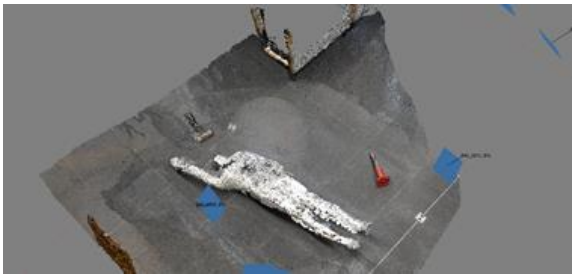
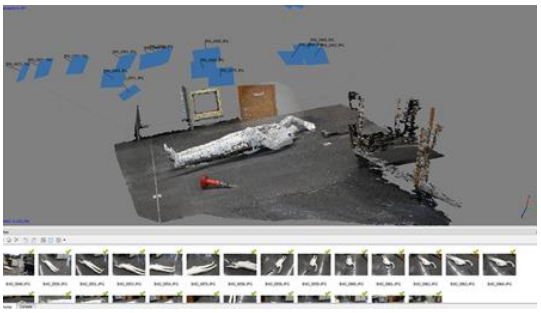
Aceleración y Paralelización del cálculo de distancias entre una nube de puntos y una superficie triangulada en 3D.





Máster Universitario en Computación en la Nube y de Altas Prestaciones

Departamento de Sistemas Informáticos y Computación

TRABAJO FINAL DE MÁSTER

Autor: Estefanía González Conchell

Director: Víctor Manuel García Mollá

Curso: 2021-2022

Resumen:

La idea principal del proyecto es acelerar, mediante computación de altas prestaciones, el cálculo de distancias entre una nube de puntos y una superficie triangulada en tres dimensiones.

Para cada punto de la nube, se desea obtener el punto de la superficie mas cercano.Para realizar ese cálculo básico se dispone como software inicial la función MATLAB de libre distribución “point2trimesh”. Con el objetivo de acelerar el cálculo se ha creado una versión en lenguaje C, compilado como un archivo “MEX”para su ejecución en MATLAB. Ese código en C se ha utilizado como base para crear una versión orientada a ordenadores multicore con la librería OpenMP, y posteriormente para crear una versión orientada a GPUs mediante la librería de CUDA. Las nuevas versiones realizan el mismo proceso,pero con una gran mejora en el tiempo de cálculo.

Esta mejora va orientada a los programas de procesamiento de nubes de puntos 3D, para conseguir una computación de alto rendimiento.

Abstract:

The main idea of the project is to speed up, through high-performance computing, the calculation of distances between a cloud of points and a three-dimensional triangulated surface.

For each point of the cloud, it is desired to obtain the closest point of the surface. To carry out this basic calculation, the freely distributed MATLAB function “point2trimesh” is available as initial software. In order to speed up the calculation, a version in C language has been created, compiled as a "MEX" file, for its execution in MATLAB. This C code has been used as the basis to create a version oriented to multicore computers with the OpenMP library, and later to create a version oriented to GPUs using the CUDA library. The new versions carry out the same process, but with a great improvement in the calculation time.

This enhancement is targeted at 3D point cloud processing programs for high performance computing.

Resum:

La idea principal del projecte és accelerar, mitjançant computació d'altes prestacions, el càlcul de distàncies entre un núvol de punts i una superfície triangulada en tres dimensions.

Per a cada punt del núvol, es vol obtenir el punt de la superfície més proper. Per realitzar aquest càlcul bàsic es disposa com a programari inicial la funció MATLAB de lliure distribució "point2trimesh". Amb l'objectiu d'accelerar el càlcul s'ha creat una versió en llenguatge C, compilat com un fitxer "MEX" per a la seva execució a MATLAB. Aquest codi en C s'ha utilitzat com a base per crear una versió orientada a ordinadors multicore amb la llibreria OpenMP, i posteriorment per crear una versió orientada a GPU mitjançant la llibreria CUDA. Les noves versions realitzen el mateix procés, però amb una gran millora en el temps de càlcul.

Aquesta millora va orientada als programes de processament de núvols de punts 3D per aconseguir una computació d'alt rendiment.

ÍNDICE GENERAL:

Índice de Figuras

Índice de tablas

Índice de listados de código

1.Introducción

Este trabajo tiene como objetivo general ofrecer una mejora que permita acelerar operaciones de alto coste de almacenamiento como puede ser una nube de puntos de un escaneo 3D, acelerar el cálculo de las distancias y puntos más cercanos desde una nube de puntos a una superficie triangulada. Se dispone de un código MATLAB [1] de libre distribución creado por Daniel Frisch, point2trimesh [2], que hace la tarea, pero es poco eficiente debido a que el lenguaje de MATLAB es un lenguaje interpretado (se compila y ejecuta a la vez).

Para el trabajo se ha estudiado el software point2trimesh. Del estudio se ha determinado que la mejor forma de acelerar el código era, en primer lugar, generar una versión escrita en forma de archivo MEX. Esto ha requerido reescribir parte del código MATLAB en "point2trimesh2" como codigo C.

Una vez reescrito el código, se ha procedido a acelerar el código resultante con diferentes técnicas:

1. Reestructurar el código para mejorar la eficiencia.
2. Paralelización mediante OpenMP.
3. Paralelización CUDA, para que el código C de los archivos MEX se pudiera convertir en Kernels de CUDA ejecutables en GPUs de la marca Nvidia.

El entorno de trabajo utilizado principalmente es MATLAB. Las plataformas computacionales han sido la maquina ASUS TUF GAMING con una RAM de 16GB con sistema operativo Windows 10 pro y MATLAB R2021a, también se ha usado por escritorio remoto la maquina Kempes con MATLAB 2020 sobre sistema operativo Linux. Esta maquina es del Instituto ITEAM con las siguientes caracteristicas, tiene un Intel Xeon E5-2697v2 con 24 cores, 22Gb de memoria y 2 GPUs tesla K20Xm, con 6 Gb de memoria cada una.

MATLAB (MATrix LABoratory) es un programa orientado al cálculo con matrices, de ahí la facilidad de trabajar con vectores y matrices. Una de las características más destacables de MATLAB es su capacidad gráfica para representar funciones en distintos sistemas de coordenadas en nuestro caso como la representación de la nube de puntos y el cálculo de distancias en coordenadas XYZ.

MATLAB ofrece un entorno de programación interactivo de alto nivel que permite expresar modelos matemáticos basados en la solución de problemas numéricos, además dispone de un conjunto de recursos adicionales (‘toolbox”) que ofrecen funciones específicas desarrolladas para explotar distintos tipos de arquitecturas de alto rendimiento tales como multicore, GPU o clúster. En este proyecto se han utilizado tarjetas gráficas desde MATLAB, con la herramienta de Parallel toolbox, instalando previamente el último driver de NVIDIA para la tarjeta gráfica.

Por otro lado, CUDA (Compute Unified Device Architecture) es el interface de programación desarrollado por NVIDIA [3] para explotar las GPUs. El desarrollo de códigos de este tipo de interfaces requiere un esfuerzo de desarrollo superior al requerido en entornos como MATLAB. Existen diversas rutinas de uso libre basadas en CUDA que pueden ser integradas en códigos C, como CUDA-X [4].

Para desarrollar los objetivos de este trabajo se ha seguido los siguientes puntos:

1. Estudio del cálculo de distancias entre un punto y una superficie triangulada en 3D del fichero point2trimesh. Funcionamiento e integración de los argumentos de entrada y salida.
2. Pruebas con casos pequeños con cálculos lineales ,vectorizados y en paralelo desde la computadora de trabajo y la maquina “Kempes” de la universidad, además de hacer uso de la computación en GPU.
3. Implementación del cálculo de distancias entre una nube de puntos y una superficie triangulada en 3D en MEX , OpenMP y finalmente CUDA.
4. Obtención de casos mas grandes con otros escaneos de la misma sala de laboratorio.
5. Evaluación comparativa del rendimiento de los diferentes métodos de computación y tiempos de ejecución.

2. Problema a resolver y software inicial

2.1. Descripción del problema de distancias de nube de puntos a superficie triangulada. Descripción de la superficie:

Una Superficie triangulada en tres dimensiones es una colección de triángulos formando una superficie. Cada triángulo se caracteriza por sus tres vértices. Por lo tanto podemos almacenar la superficie con dos matrices:

1. Vertices (Nv,3): (números reales) donde Nv es el número de vértices de la superficie triangulada. En esta matriz guardamos las coordenadas de los vértices, en cada fila guardamos las coordenadas de un vértice.
2. Faces (Nt,3): donde Nt es el número de triángulos. En cada fila de esta matriz guardamos los índices de tres vértices. Es decir, si la fila x de la matriz Faces contiene los números i,j,k, entonces las coordenadas de los vértices del triángulo x se encuentran en las filas i,j,k de la matriz Vertices.

A continuación se muestra el ejemplo gráfico de la versión sencilla del problema, con cuatro triángulos, seis vértices y tres puntos de consulta;

Listado 2.1: Argumentos de entrada ejemplo original

```
FV.faces    = [5 3 1; 3 2 1; 3 4 2; 4 6 2];
FV.vertices = [2.5 8.0 1; 6.5 8.0 2; 2.5 5.0 1; 6.5 5.0 0; 1.0 6.5 1; 8.0 6.5 1.5];
points      = [2 4 2; 4 6 2; 5 6 2];
```

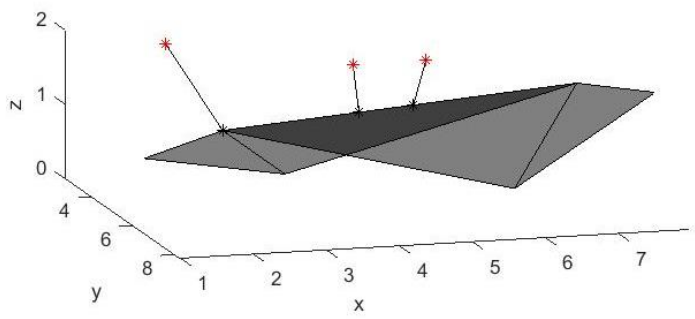


Figura 2.1: Resultado del ejemplo original de “point2trimesh”

Cada línea entre dos vértices de un triángulo es una arista.

Dada una nube de puntos con Np puntos, para cada punto P de la nube, se pretende calcular el punto en la superficie triangulada (en un vértice, o en una arista, o en la superficie de uno de los triángulos) con menor distancia al punto P.

La distancia de un punto a la superficie se obtiene realizando tres cálculos, para cada punto de la nube:

-Cálculo de la menor distancia desde el punto a uno de todos los vértices de la superficie y obtención de ese punto de menor distancia a un vértice.

-Cálculo de la menor distancia desde el punto a una de las aristas de la superficie y obtención de ese punto de menor distancia a una arista.

- Cálculo de la menor distancia desde el punto a cada una de las superficies de triángulos y obtención de ese punto a menor distancia.

Luego el punto de mínima distancia será el mínimo de los tres y se devuelve el punto correspondiente.

Este proceso se repite para todos los puntos de la nube.

2.2. Descripción de la función MATLAB “point2trimesh”

Dada una nube de puntos y una superficie triangulada 3D, La función “point2trimesh” calcula las líneas más cortas que conectan cada punto con la triangulación en 3D. Se devuelven los puntos más cercanos a la superficie, así como las distancias.

La distancia se asigna de acuerdo a las normales de la cara para identificar en qué lado de la superficie reside el punto de consulta. Dentro de point2trimesh hay varios algoritmos implementados, que se pueden seleccionar en los argumentos de entrada. Los que hemos usado han sido las versiones llamadas “linear”, versión secuencial que va punto a punto, la versión “vectorized” que vectoriza los cálculos de distancias pero que no es más rápida que la secuencial y la versión “parallel” que hace uso de la instrucción parfor del Parallel toolbox para usar varios cores a la vez, hasta 12.

Existen otras posibilidades pero la experimentación ha demostrado que son más lentas y no son de interés para este trabajo, ya que no está optimizado para la velocidad del cálculo que es nuestro objetivo principal.

El algoritmo va de punto a punto: para cada punto, calcula;
- El vértice más cercano
- El punto más cercano en los bordes
- El punto más cercano en las superficies del triángulo

Se calcula y se devuelve la distancia mínima de estos tres.

Argumentos de entrada:

Para poder definir la superficie triangulada e introducir los datos, la estructura de los “arrays” se compone en entradas separadas, donde cada nombre de campo corresponde a un nombre de parámetro de entrada. De los cuales se definen a continuación; caras, vértices y puntos.

Parámetros:
- *'Faces'* : (#faces x 3) Representa la conectividad de la triangulación. Cada fila define una cara, los 3 elementos son los ID de los vértices.
- *'Vertices'* : (#Vertices x 3) Es una matriz de puntos. Las columnas son coordenadas x, y, z, los números de la fila son los ID de los vértices.
-*'QueryPoints'*: (#qPoints x 3) También es una matriz de puntos. Las columnas son coordenadas x, y, z. Cada fila define un punto de consulta. Puede estar vacío.

- 'Algorithm':
  - 'linear': Se utiliza cuando hay pocos puntos. Los puntos de consulta se procesan sucesivamente en un bucle "for".
  - 'parallel': Se utiliza cuando la cantidad de puntos es suficientemente grande. Los puntos de consulta se procesan en paralelo con "parfor". Una ventaja de MATLAB es que si no existe ningun grupo paralelo, lo crea automaticamente. Y se apaga después de 30 minutos de inactividad. Además se puede cambiar el "Numero de *Workers*", si MATLAB no los reconoce se puede cambiar dentro de administrar perfiles de clúster.
  - 'vectorized': Esta opción es la menos recomendada, está para mostrar que la vectorización no siempre acelera el cálculo. Los puntos de consulta se procesan en conjunto de manera vectorizada. Hay que considerar el tamaño de la RAM si tiene muchos puntos.

Argumentos de salida:

- *distances:* es un vector de distancia. Distancia de punto a superficie. El signo depende de los vectores normales.
- *surface_points:* es una matriz de puntos. Los puntos correspondientes más cercanos en la superficie.

3.Programación

El algoritmo básico en la función point2trimesh tiene un bucle que recorre todos los puntos como se ha descrito anteriormente, toda la nube punto a punto y para cada uno de los puntos llama a tres funciones;

1. *distance_to_vertices*

2. *distance_to_edges*

3. *distance_to_surfaces*

Cada una de estas funciones MATLAB realiza respectivamente el cálculo de menor distancia para vértices, aristas y superficie de triángulos. Por último, se compara las tres distancias obtenidas y se escoge la mínima y se obtiene la cara, vértice, arista que da la distancia mínima. De esta forma se obtiene el punto de la malla más cercano al punto.

Listado 3: Pseudocódigo de algoritmo básico

```
Input: Faces, Vertices, queryPoints, Np, Nf, NV
       For i=1: Np //procesamos el punto querypoints(i)
           [Dv,Pv]=Distance_to_vertices(Faces,Vertices, querypoints(i));
           [De,Pe]=Distance_to_edges(Faces,Vertices, querypoints(i));
           [Ds,Ps]=Distance_to_surface(Faces,Vertices, querypoints(i));
            Dp=min(Dv, De, Ds)
            If (Dp==Dv)
                Pout=Pv
            Else if (Dp==De)
                Pout=Pe
            Else
                Pout=Ps;
            End If
         End For
```

Este es el funcionamiento general de problema pero también hemos comprobado las distintas versiones originales de point2trimesh para hacer la comparativa de los algoritmos de cálculo que se pueden elegir en los argumentos de entrada, de los cinco algoritmos, se ha hecho hincapié en el lineal, el paralelo y el vectorizado ya que con estos tres tanto con un caso pequeño como más grande podemos visualizar las diferencias en tiempo de ejecución y tamaño.

El cálculo punto a punto de todas las distancias no es eficiente. Por eso vamos a utilizar otras implementaciones para acelerar el cálculo.

Otra forma de mejorar los tiempos de computo y que se ha notado con los resultados es utilizar los procesadores gráficos, GPU, para realizar las ejecuciones de cálculo, en lugar de los habituales procesadores genéricos, CPU. Si se tiene disponible el Parallel toolbox se puede utilizar la GPU desde MATLAB.

Para poder usar la GPU primero hay que enviar todos los datos que necesita el problema. La función que se usa en MATLAB para enviar los argumentos de entrada es“gpuArray”, la única restricción en tener en cuenta es que las matrices que se envíen deben ser densas. Lo que se haga con esos datos se hará directamente en la GPU. Otra forma de computación en GPU desde MATLAB es programar “Kernels” usando CUDA+C y llamarlo desde MATLAB que se verá mas adelante.

Como ya se ha mencionado, point2trimesh tiene varias versiones de algoritmo, una paralelizada (con parfor, haciendo uso del parallel Toolbox de MATLAB) y otra vectorizada. Esas versiones son más rápidas que la versión básica, pero también están escritas en lenguaje MATLAB, lo que les resta velocidad. Además la estructura (hacer un bucle y para cada punto hacer tres llamadas a funciones) no es ni mucho menos ideal desde el punto de vista de la eficacia. Mucho más razonable es modificar las funciones Distance_to_vertices, Distance_to_edges, y Distance_to surfaces, de forma que una sola llamada realicen el cálculo para todos los puntos. Además, al realizar el cálculo de esta forma y escribir las funciones en forma de archivo MEX se facilita la posterior paralelización, haciendo que los cálculos de varios puntos se realicen en paralelo.

Vamos a detallar el proceso de generación para las diferentes versiones. El proceso que se sigue es muy similar a las tres funciones. Usaremos distance_to_vertices a modo de ejemplo, los códigos correspondientes para distance_to_edges y para distance_to_surfaces pueden verse en los apéndices.

Listado 3.1: Distance_to_vertices original MATLAB

```
function [D,P,F,V] = distance_to_vertices(faces,vertices,qPoint,normals)

% find nearest vertex
[D,nearestVertexID] = min(sum(bsxfun(@minus,vertices,qPoint).^2,2),[],1);
D = sqrt(D);
P = vertices(nearestVertexID,:); % (1 x 3)
V = nearestVertexID;

% find faces that belong to the vertex
connectedFaces = find(any(faces==nearestVertexID,2)); % (#connectedFaces x 1) face
indices
assert(length(connectedFaces)>=1,'Vertex %u is not connected to any
face.',nearestVertexID)
F = connectedFaces(1);
n = normals(connectedFaces,:); % (#connectedFaces x 3) normal vectors

% scalar product between distance vector and normal vectors
coefficients = dot2(n,qPoint-P);
sgn = signOfLargest(coefficients);
D = D*sgn;

end

```

En éste listado 3.1 , podemos observar la función distance_to_vertice de la versión original de “point2trimesh”, que lo usaremos de referencia para el resto de implementaciones.

4.Tecnologias utilizadas:

4.1.Archivos MEX

Los archivos MEX [7] proceden de ejecutables MATLAB y son funciones que se pueden llamar desde MATLAB escritas en C y C++. La finalidad de estos archivos está en que con ellos es posible proporcionar velocidades de ejecución más alta y evitar los cuellos de botella en las aplicaciones. Cuando se compila los ficheros MEX, se crea una función envoltorio que permite que sean pasados y devueltos tipos de datos de MATLAB. De este modo, desde MATLAB se pueden cargar de forma dinámica y ser invocados códigos en C o C++ como si fuesen funciones integradas del propio MATLAB. Para apoyar el desarrollo de los archivos MEX, MATLAB ofrece funciones de interfaz que facilitan la transparencia de datos entre archivos MEX y MATLAB.

Para crear un archivo MEX se necesita:

1. Tener un código fuente escrito en C o C++.

2. Un compilador que sea soportado por MATLAB.

3. La API de C y C++ para manejar la estructura de datos de MATLAB (*mxArray*) y la librería MEX para realizar operaciones del entorno de MATLAB desde el código C.

Lo aconsejable es completar la estructura estándar de un fichero MEX. La rutina debe contener el código implementado del MEX. Dónde se especifica el número de entradas y salidas desde la interfaz de MATLAB. En la función MEX las estructuras de datos que se conectan en los dos entornos son identificados por *prhs, plhs, nrhs, nlhs*, donde *plhs* es un array de salida de argumentos y *nlhs* es el número de salidas de los argumentos, y los otros dos son el array y el número de entradas de los argumentos, como se observa en el listado 4.1.1.

La ventaja de usar los ficheros MEX es la reducción de latencia en el envío y la recepción de datos. La latencia del envío de datos a las rutinas invocadas es mínima en cambio la recepción de los datos es más costosa si son tamaños de datos grandes.

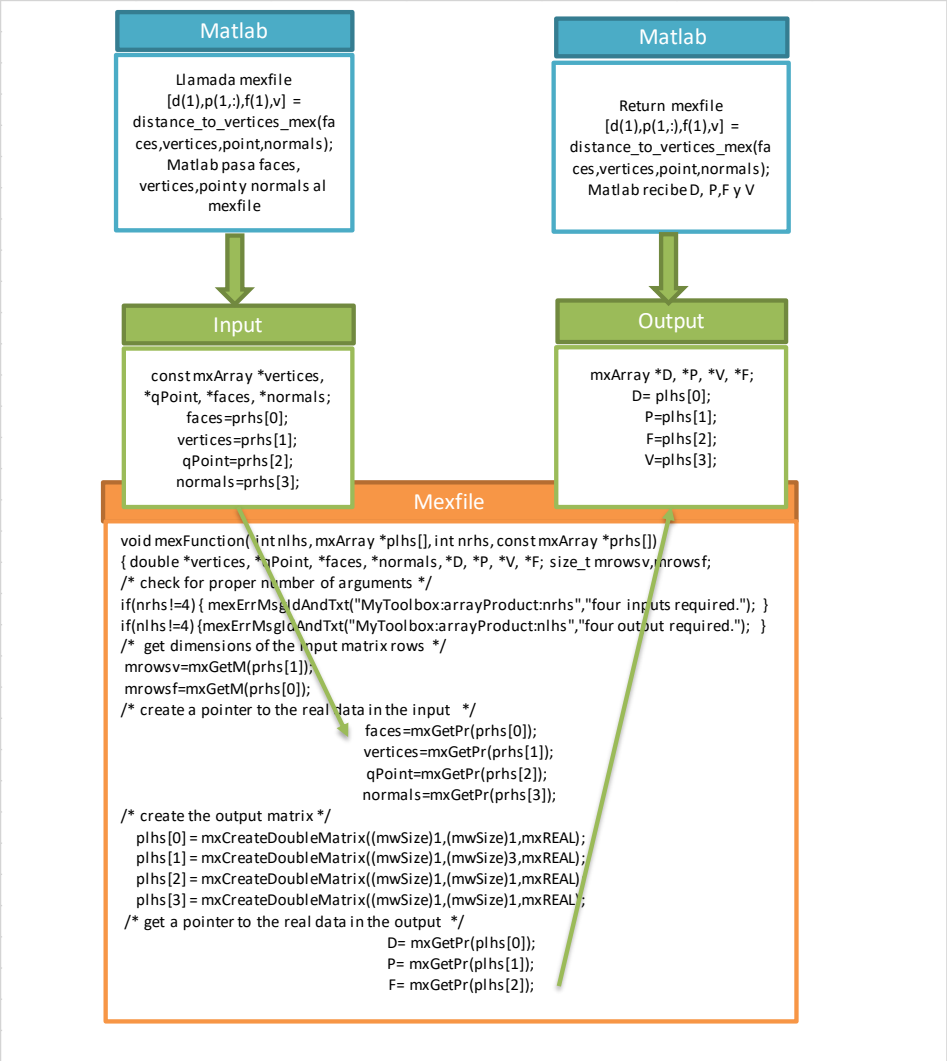


Figura 4.1: Entradas y salidas de datos a través de archivo distance_to_vertices_mex.c

En la Figura 4.1 se muestra el ejemplo del paso de parámetros a través del fichero MEX de *distance_to_vertices*, con la llamada de la función en "point2trimesh". Donde las variables de entrada procedentes de MATLAB son *faces, vertices, qPoint y normals* que son de tipo *mxArray* [9] y se accede a la matriz de tipo *double* (que son los argumentos de entrada) con la función *mxGetPr*. Y de la misma forma son devueltos los argumentos de salida de tipo *mxArray* con la función *mxGetPr*. La numeración de los *prhs[]* indica el orden de los parámetros de entrada y los *plhs[]* indica el orden de los parámetros de salida hacia MATLAB. Además, de la función *mxGetPr* también se ha utilizado la función *mxGetM* para que me devuelva el número de filas de la matriz específica y así obtener la dimensión de la matriz de entrada. Y la función *mxCreateDoubleMatrix* para crear un *mxArray m-by-n* para crear la salida de la matriz específica.

4.1.1. Programación con MEX

Volviendo al problema el primer paso para acelerar el cálculo de las distancias, como ya se ha comentado anteriormente, es reestructurar el código para mejorar la eficiencia, sustituyendo el bucle por tres archivos MEX mediante un bucle en C; Uno que calcule las distancias de todos los puntos a todas las caras, otro para las aristas y otro para vértices. Entonces la estructura se quedaría así:

1.1) Archivo MEX, *distance_to_surfaces_mex*, que realiza el cálculo de la distancia a todas las caras, se queda con la distancia mínima y la cara correspondiente.
1.2) Archivo MEX, *distance_to_vertices_mex,* que realiza el cálculo de la distancia a todos los vértices, se queda con la distancia mínima y el vértice correspondiente.
1.3) Archivo MEX, distance_to_edges_mex, que realiza el cálculo de la distancia a todas las aristas, se queda con la distancia mínima y la arista correspondiente.
1.4) Por último, para cada punto se obtiene la distancia mínima y la cara, vértice, arista correspondiente.

La parte complicada es transformar el código MATLAB vectorizado, en código C. Como resultado se obtiene tres archivos MEX (compilados con -O3 en G). Al igual que en la parte de programación usaremos *distance_to_vertices_mex* a modo de ejemplo, los códigos correspondientes para *distance_to_edges_mex* y para *distance_to_surfaces_mex* pueden verse en los apéndices.

Los archivos MEX se componen en dos partes, está la parte computacional y la parte de paso de parámetros. Como se puede observar la parte computacional serían los cuadros 4.1.1 y 4.1.2 donde se encuentra la función de *distance_to_vertices_mex* y la parte de paso de parámetros correspondería el ultimo cuadro 4.1.3 desde la función de *mexFunction* [8]. En esta función se controla el paso de parámetros de entrada y de salida de MATLAB al código C. Y la función de la parte computacional es prácticamente la traducción de la función de MATLAB, con la diferencia de que tiene un bucle externo que recorre todos los puntos en una sola llamada.

Listado 4.1.1: Distance_to_vertices_mex

```
#include <math.h>
#include <stdio.h>
#include "mex.h"

void distance_to_vertices_mex(double *faces, double *vertices, double *qPoint,
double *normals, double *D, double *P,double *F, double *V, mwSize mrowsv ,mwSize
mrowsf, mwSize npoints )
{
    mwSize ip;

    for (ip=0; ip<npoints; ip++) {

        mwSize i, j, imind;
        double mind,coefficients,coef0,coef1,coef2,auxd;
        int find,sgn,coeflargest;


        mind=mxGetInf();
        imind=0;
     for (i=0; i<mrowsv; i++)
       {
        auxd=0;

        for(j=0; j<3; j++){
        auxd = auxd + pow(vertices[i+j*mrowsv]- qPoint[ip +j*npoints],2);}

        if (auxd< mind)
           {
            mind=auxd;
            imind=i;
             }
      }


        D[ip] = sqrt(mind);
        V[ip] = imind+1;
        P[ip]= vertices[imind];
        P[ip+npoints]= vertices[imind+1*mrowsv];
        P[ip+2*npoints]= vertices[imind+2*mrowsv];

        /**/
```

Listado 4.1.2: Distance_to_vertices_mex

```
            find =0;
            coeflargest=0;


      for (i=0; i<mrowsf; i++)

       {

           for(j=0; j<3; j++){

           if (faces[i+j*mrowsf]==imind){


               if ( find==0) {
                   {F[ip] = i + 1;
                   find = 1;
               }

               coef0=0;
               coef1=0;
               coef2=0;
               coef0+=normals[i]*(qPoint[ip]- P[ip]);
               coef1+=normals[i+1*mrowsf]*(qPoint[ip+npoints]- P[ip+npoints]);
               coef2+=normals[i+2*mrowsf]*(qPoint[ip+2*npoints]- P[ip+2*npoints]);
               coefficients=coef0+coef1+coef2;
               }

           if (abs(coefficients)> abs(coeflargest))
              {
               coeflargest=coefficients;
               }
    }
    }
           if ( coeflargest>= 0){
           sgn=1;}
           else {
           sgn=-1;}

         D[ip]=(D[ip])*sgn;
  }
  }
  }
```

Listado 4.1.3: Distance_to_vertices_mex

```
86  void mexFunction( int nlhs, mxArray *plhs[],
87                    int nrhs, const mxArray *prhs[])
88  {
89      double *vertices, *qPoint, *faces, *normals, *D, *P, *V, *F;
90      size_t mrowsv,mrowsf, npoints;
91      /* check for proper number of arguments */
92      if(nrhs!=4) {
93          mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs","four inputs required.");
94      }
95      if(nlhs!=4) {
96          mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs","four output required.");
97      }
98
99      /*  get dimensions of the input matrix rows  */
100
101     mrowsv=mxGetM(prhs[1]);
102     mrowsf=mxGetM(prhs[0]);
103     npoints=mxGetM(prhs[2]);
104
105     /* get dimensions of the input matrix cols*/
106     /*ncols = mxGetN(prhs[0]);
107
108     /* create a pointer to the real data in the input   */
109
110     faces=mxGetPr(prhs[0]);
111     vertices=mxGetPr(prhs[1]);
112     qPoint=mxGetPr(prhs[2]);
113     normals=mxGetPr(prhs[3]);
114
115     /* create the output matrix */
116     plhs[0] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)1,mxREAL);
117     plhs[1] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)3,mxREAL);
118     plhs[2] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)1,mxREAL);
119     plhs[3] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)1,mxREAL);
120
121     /* get a pointer to the real data in the output  */
122      D= mxGetPr(plhs[0]);
123      P= mxGetPr(plhs[1]);
124      F= mxGetPr(plhs[2]);
125      V= mxGetPr(plhs[3]);
126
127     /* call the computational routine */
128    distance_to_vertices_mex(faces, vertices, qPoint,
129    normals,D,P,F,V,mrowsv,mrowsf,npoints);
130 }
131
132
133
```

4.2.OpenMP

OpenMP [6] es una API (Application Programming Interface) que puede ser utilizada para diseñar programas paralelos multithread en memoria compartida. Es decir, un programa OpenMP es un programa secuencial con directivas OpenMP que pueden ser usadas con FORTRAN, C, C++. La principal ventaja de OpenMP es su facilidad de uso ya que permite crear concurrencia, sincronización y tratar los datos sin necesidad de tener que preocuparnos por compartir el mismo recurso en un programa de varios subprocesos mutexes (mutual exclusión object), variables de condición y otros aspectos de bajo nivel.

OpenMP está compuesto de tres elementos:

1. Las directivas de compilación

2. Las rutinas de librería en tiempo de ejecución

3. Las variables de entorno

Hay muchos compiladores que soportan diferentes versiones de OpenMP. Como se muestra en la siguiente lista, Tabla 4.2. En general para compilar una aplicación con soporte OpenMP y, sí se usa la librería, sólo necesita agregar un indicador de compilación y su encabezado <omp.h>. El más conocido es el compilador GCC.

Lista de compiladores OpenMP
GCC (incluidos gcc, g ++ y gfortran): -fopenmp
LLVM: -fopenmp
Compilador-suite de Intel (incluidos icc, icpc e ifort): -qopenmp (y -fopenmp para compatibilidad con GCC / LLVM)
IBM XL compilador-suite (incluyendo xlc, xlC y xlf): -xlsmp=omp
PGI compiler-suite (incluyendo pgcc pgc ++ pgfortran): '-mp'

Tabla 4.2. Lista de compiladores OpenMP

Las rutinas de librería en tiempo de ejecución de OpenMP vienen dadas por el encabezado <omp.h>, donde se declara dos tipos, funciones que se pueden usar para controlar y consultar el entorno de ejecución en paralelo, y funciones de bloqueo que se pueden usar para sincronizar el acceso a los datos. Pero en nuestro caso, no se ha añadido la librería porque no se ha usado ninguna de sus funciones, lo que nos interesaba es la utilización de OpenMP para crear hilos con los que intentar solapar a lo más posible el cómputo de las transmisiones de mensajes y escritura a disco.

Están incluidas las siguientes variables de entorno en la API de C y C++ de OpenMP, la Tabla 4.2.1, que controlan la ejecución de código paralelo. Hay que tener en cuenta, que las variables de entorno deben estar en mayúscula y se omiten las modificaciones en los valores una vez iniciado el programa.

Variable de entorno	Descripción
OMP_SCHEDULE	Modifica el comportamiento de la cláusula schedule cuando se especifica en una directiva `for` o `parallel for` .
OMP_NUM_THREADS	Establece el número máximo de subprocesos en la región paralela, a menos que lo invalide omp_set_num_threads o num_threads.
OMP_DYNAMIC	Especifica si el tiempo de ejecución de OpenMP puede ajustar el número de subprocesos en una región paralela.
OMP_NESTED	Especifica si el paralelismo anidado está habilitado, a menos que el paralelismo anidado esté habilitado o deshabilitado con `omp_set_nested`.

Tabla 4.2.1. Variables de entorno de OpenMP

El modelo de programación en OpenMP hace uso de que un proceso puede consistir en múltiples threads de ejecución en la máquina de memoria compartida. Utiliza el modelo Fork-Join y el programador tiene control explícito de la paralelización. Se supone que el programa tiene secciones de código secuencial y secciones paralelizables y que el programador interviene sobre secciones paralelizables. OpenMP hace de intermediario entre su código y su ejecución. Cada instrucción #pragma omp se convierte en llamadas a su función correspondiente de la biblioteca OpenMP. La ejecución de subprocesos múltiples siempre es manejada por el sistema operativo (OS).

4.2.1. Programación con OpenMP

El segundo paso que se ha llevado a cabo para acelerar el cálculo de las tres distancias es la paralelización con OpenMP. Sencillamente usa el pragma para paralelizar el bucle externo, el de los puntos. Y compila adecuadamente los Mex resultantes. Haciendo referencia al cuadro anterior 4.1.1 de la función distance_to_vertices_mex sería paralelizar el bucle de la línea 8 a la 85 con el "pragma parallel for" como se muestra en el siguiente cuadro 4.2.1. en las líneas 13 y 14.

Listado 4.2.1: Distance_to_vertices_omp

```
1   #include <math.h>
2   #include <stdio.h>
3   #include "mex.h"
4
5   void distance_to_vertices_omp(double *faces, double *vertices, double *qPoint,
6   double *normals, double *D, double *P,double *F, double *V, mwSize mrowsv ,mwSize
7   mrowsf,mwSize npoints )
8   {
9
10      mwSize ip;
11
12
13      #pragma omp parallel{
14      #pragma omp for
15
16      for (ip=0; ip<npoints; ip++) {
17
18           mwSize i, j, imind;
19           double mind,coefficients,coef0,coef1,coef2,auxd;
20           int find,sgn,coeflargest;
21
22
23           mind=mxGetInf();
24           imind=0;
25
26      for (i=0; i<mrowsv; i++)
27       {
28          auxd=0;
29
30           for(j=0; j<3; j++){
31           auxd = auxd + pow(vertices[i+j*mrowsv]- qPoint[ip +j*npoints],2);}
32
33           if (auxd< mind)
34              {
35               mind=auxd;
36               imind=i;
37               }
38       }
```

Listado 4.2.2: Distance_to_vertices_omp

```
            D[ip] = sqrt(mind);
            V[ip] = imind+1;
            P[ip]= vertices[imind];
            P[ip+npoints]= vertices[imind+1*mrowsv];
            P[ip+2*npoints]= vertices[imind+2*mrowsv];

            /**/
            find =0;
            coeflargest=0;


      for (i=0; i<mrowsf; i++)

       {

            for(j=0; j<3; j++){

            if (faces[i+j*mrowsf]==imind){


                if ( find==0) {
                    {F[ip] = i + 1;
                    find = 1;
                }

                coef0=0;
                coef1=0;
                coef2=0;
                coef0+=normals[i]*(qPoint[ip]- P[ip]);
                coef1+=normals[i+1*mrowsf]*(qPoint[ip+npoints]- P[ip+npoints]);
                coef2+=normals[i+2*mrowsf]*(qPoint[ip+2*npoints]- P[ip+2*npoints]);
                coefficients=coef0+coef1+coef2;
                }

            if (abs(coefficients)> abs(coeflargest))
               {
                coeflargest=coefficients;
                }
      }
    }
            if ( coeflargest>= 0){
            sgn=1;}
            else {
            sgn=-1;}

    D[ip]=(D[ip])*sgn;
  }
  }
  }

```

Listado 4.2.3: Distance_to_vertices_omp

```
90  void mexFunction( int nlhs, mxArray *plhs[],
91                    int nrhs, const mxArray *prhs[])
92  {
93      double *vertices, *qPoint, *faces, *normals, *D, *P, *V, *F;
94      size_t mrowsv,mrowsf,npoints;
95      /* check for proper number of arguments */
96      if(nrhs!=4) {
97          mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs","four inputs required.");
98      }
99      if(nlhs!=4) {
100         mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs","four output required.");
101     }
102
103     /*  get dimensions of the input matrix rows  */
104
105     mrowsv=mxGetM(prhs[1]);
106     mrowsf=mxGetM(prhs[0]);
107     npoints=mxGetM(prhs[2]);
108
109
110     /* get dimensions of the input matrix cols*/
111     /*ncols = mxGetN(prhs[0]);
112
113     /* create a pointer to the real data in the input   */
114
115     faces=mxGetPr(prhs[0]);
116     vertices=mxGetPr(prhs[1]);
117     qPoint=mxGetPr(prhs[2]);
118     normals=mxGetPr(prhs[3]);
119
120     /* create the output matrix */
121     plhs[0] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)1,mxREAL);
122     plhs[1] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)3,mxREAL);
123     plhs[2] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)1,mxREAL);
124     plhs[3] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)1,mxREAL);
125
126     /* get a pointer to the real data in the output  */
127      D= mxGetPr(plhs[0]);
128      P= mxGetPr(plhs[1]);
129      F= mxGetPr(plhs[2]);
130      V= mxGetPr(plhs[3]);
131
132     /* call the computational routine */
133    distance_to_vertices_omp(faces, vertices, qPoint,
134    normals,D,P,F,V,mrowsv,mrowsf,npoints);
135 }
136
137
```

Por lo tanto, el formato sería igual al fichero Mex, pero solo con la parte computacional en paralelo y la parte del paso de parámetros se mantendría en secuencial.

También hay que tener en cuenta que la compilación es como si fuera un fichero Mex, pero con OpenMP, solo tendríamos que introducir la ruta de optimización -02 con ‘’OPTIMFLAGS ‘’ dentro de la compilación Mex y llamar a la función c.

4.3. GPUs

La existencia de los conjuntos de tecnologías de procesamiento masivo de datos en la actualidad, son de gran ayuda a a la hora de automatizar la resolución de cualquier operación.

Sabemos que la solución a esto, puede tardar un prolongado tiempo y su mejora se puede realizar mediante diferentes modelos de programación y la utilización de herramientas tecnológicas como en este caso las GPUs.

Las GPUS (Graphics processing units) son un componente interno que ayuda a aligerar la carga de información que se envia a procesar dentro de la CPU y asi mejorar la eficiencia y rendimiento de la computadora a la hora de procesar datos gráficos masivos.

Una unidad de procesamiento gráfico la componen millares de núcleos de tamaño mas pequeños y eficientes, diseñados para el rendimiento en paralelo, mientras que la CPU la componen varios nucleos optimizados para el procesamiento en serie.

Otro aspecto a tener en cuenta, es el ancho de banda. Las GPUs tienen un ancho de banda muy superior a las CPUs, esto es debido a que las CPUs son procesadores de propósito general que tienen que satisfacer los requerimientos de diversos tipos de aplicaciones, sistemas operativos y dispositivos de entrada-salida, por lo que disponen de menor espacio para poder realizar cálculos y de ahí que su rendimiento para el cálculo sea menor. Mientras que las GPUs destina la mayor parte del espacio a unidades de cálculo.

Esta ventaja de las GPUs va más ligado a la industria del videojuego donde necesitan una mayor capacidad de cómputo porque deben ser capaces de realizar operaciones en como flotante de forma masiva. Por esto, las GPUs destinan más espacio a las unidades de cómputo y menos a la lógica de control. Necesitan menor tamaño de lógica de control porque integra un sistema de gestión que controla qué datos están listos para uso y cuáles no.

Por eso, desde hace tiempo el modelo computacional CPU-GPU está muy extendido en el ámbito científico, porque aprovecha el control lógico de la CPU y la potencia de cálculo de la GPU.

En la programación paralela referida en este proyecto y en proyectos en los que se utilizan datos gráficos, estas unidades sirven de gran ayuda para que el procesador central no se sobrecargue. Como son los experimentos de cómputos matemáticos pesados usando ordenes puramente gráficas. A pesar de la dificultad de programación ofrecida por esta vía se comprobó la potencialidad que tienen las GPU para el computo númerico y fruto de ello NVIDIA lanzó su lenguaje de programación propietario denominado Cg.

4.4. CUDA

CUDA (Compute Unified Device Architecture) es una plataforma de computación paralela y un modelo de programación creado por Nvidia que permite el uso de varios lenguajes de programacion C , C++, Fortran, y gestionar o codificar algoritmos en la GPU.

Esta arquitectura de calculo en paralelo, está compuesta por una jerarquía de grupo de hilos, un grupo de barreras de sincronización y un modelo de recursos compartidos (memorias compartidas).

Estas características permiten al programador en cuestión, dividir el problema en subproblemas secundarios resueltos de forma paralela y a su vez esos, en programas más pequeños. Está diseñado para que se creen ciertas aplicaciones mediante una forma transparente escalen su paralelismo y así poder incrementar el número de núcleos computacionales.

Este modelo de programacion presenta ciertas ventajas sobre otros tipos de computación sobre GPU frente a las CPU:

-Se adapta a otras arquitecturas sin necesidad de reescribir todo el código

- Posee un entorno de desarrollo diseñado para proporcionar una baja o minima curva de aprendizaje para los programadores familiarizados con el lenguaje de programación estándar C.

- Permite el procesamiento tanto gráfico como no gráfico.

-Otras como la lecturas dispersas y rapidas o memorias compartidas

También presenta una serie de limitaciones como pueden ser:

-La utilización de punteros a función o variables estáticas dentro de funciones o funciones con numero de parámetros variable

-No soporta números desnormalizados o NaNs en precisión simple ni renderizado de texturas.

-Los hilos han de lanzarse en grupos de 32, con miles de hilos en total por razones de eficiencia.

4.4.1 Estructura de un programa CUDA

La estructura del programa CUDA está relacionada con la arquitectura y el modelo computacional de las tarjetas NVIDIA aptas para CUDA que corresponden a un multiprocesador, como se ilustra en la Figura 4.4.1. En esta Figura se puede apreciar la arquitectura de la tarjeta NVIDIA con 8 multiprocesadores, con 16 núcleos cada uno y sus respectivas unidades de memoria. El mecanismo de control de la arquitectura CUDA corresponde al de las computadoras tipo SIMD, ya que la programación en paralelo CUDA consiste en ejecutar un conjunto de hilos que realizan las mismas operaciones sobre múltiples datos.

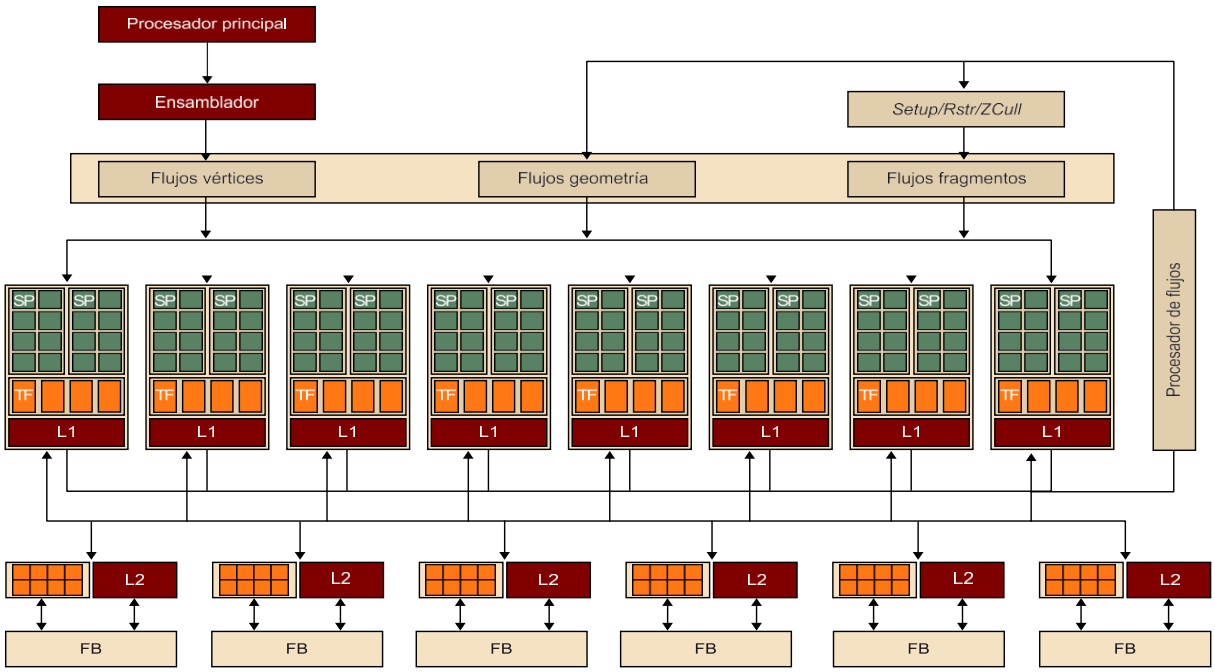


Figura 4.4.1: Arquitectura de Tarjeta NVIDIA con 128 núcleos

Por otro lado, CUDA usa la memoria compartida como medio de comunicación entre multiprocesadores (módulos L1 y L2 de la Figura 4.4.1) que componen una tarjeta apta para esta arquitectura.

El modelo de programación CUDA supone que los hilos se ejecuten en una unidad física distinta, que actúa como coprocesador (*device*) al procesador *(host*) donde se ejecuta el programa (Figura 4.4.1.1). Para esta estructura, el programador prepara todos los datos que necesitará la GPU, debe inicializar con las reservas de memoria en el *host* y en el *device* realizar un traspaso de memoria del *host* al *device* o viceversa. Una estructuración correcta del programa evitaría funcionamientos incorrectos por problemas de memoria o rendimiento. Es importante que el programador conozca su arquitectura para poder sacar el máximo rendimiento de la GPU.

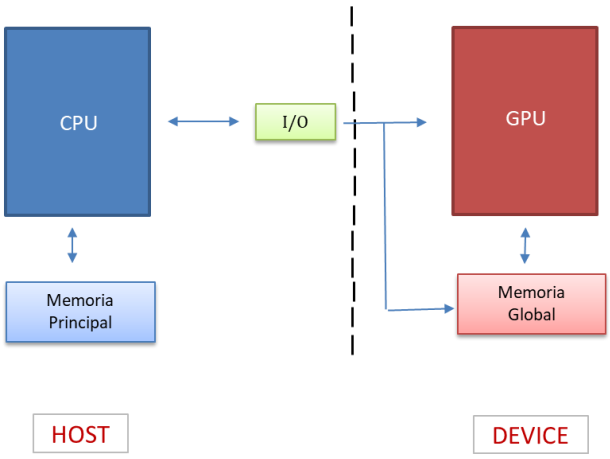


Figura 4.4.1.1: Arquitectura CPU-GPU

Hay que tener en cuenta que el *kernel* es invocado desde un lenguaje de alto nivel. Esto implica que hay que asegurarse que la integridad de los datos no se altere durante todo el proceso. El lenguaje de programación es C, donde las llamadas *kernels* son funciones C que se ejecutan en paralelo por *N hilos* diferentes. Los *kernels*, se ejecutan de forma secuencial en el *device*.

4.4.2. Kernels CUDA

El código que se ejecuta en el dispositivo (*Kernel*) es la función que ejecutan los diferentes flujos durante la fase paralela, cada uno en el rango de datos que corresponde. CUDA sigue el modelo SPMD (single-program multiple-data), es decir, todos los flujos ejecutan el mismo código. El código siguiente muestra el *Kernel* de una de las distancias del cálculo del problema, en este caso hemos dado ejemplo con distance_to_vertices.

Listado 4.4.2: Kernel de distance_vertices

```
1   void __global__ distance_vertices_CUDA(const double *qPoint,
2                                          const double *faces,
3                                          const double *vertices,
4                                          const double *normals,
5                                          int npoints,
6                                          int mrowsv,
7                                          int mrowsf,
8                                          double *D,
9                                          double *P,
10                                         double *F,
11                                         double *V
12                                         )
13  {
14   int ip=threadIdx.x+blockDim.x*blockIdx.x;
15
16       if (ip<npoints)
17
18  { …
```

Podemos apreciar la utilización de la palabra clave específica de CUDA **`__global__`** ante la declaración de **`distance_vertices_CUDA`**. Esta palabra clave indica que esta función es un *kernel* y que hay que llamarlo desde el procesador principal para generar el *grid* de flujos que ejecutará el *kernel* en el dispositivo, en este caso lo hemos llamado con **`parallel.gpu.CUDAKernel`** desde el programa de "point2trimesh" del cual hemos modificado y nombrado "point2trimeshCUDA".

En CUDA [5] un *kernel* se ejecuta mediante un conjunto de flujos (como puede ser un vector o una matriz de flujos). Como todos los flujos se ejecutan en el mismo *kernel,* se necesita un mecanismo que permita diferenciarlos y así poder asignar la parte correspondiente de los datos a cada flujo (por ejemplo, **`threadIdx.x y threadIDx.y`** si tenemos en cuenta dos dimensiones).

Normalmente, los *grids* que se utilizan en CUDA están formados por muchos flujos. Un *grid* consiste en unos o más bloques de flujos. Todos los bloques de un *grid* tienen el mismo número de flujos y deben estar organizados del mismo modo. Es decir, el índice de un hilo y su ID de subprocesos se relacionan entre sí de una manera directa: en este caso para un bloque unidimensional, son los mismos. Con el ejemplo anterior vemos que hay un flujo de ejecución en una dimensión (**`threadIdx.x`**).

Hay un límite para el número de flujos por bloque, ya que se espera que todos los flujos de un bloque residan en el mismo núcleo del procesador además de compartir la memoria de ese núcleo, del cual es limitada. En las GPU de hoy en día, un bloque puede contener hasta 1024 flujos.

Sin embargo, un *kernel* puede ser ejecutado por múltiples bloques de flujos. Por lo que el número total de flujos es igual al número de flujos por bloque multiplicado por el número de bloques.

Cada bloque de un *grids* se organiza en bloques de *threads* como se muestra en la siguiente imagen la Figura 4.4.2. Cada bloque dentro del *grid* tiene una coordenada única en un espacio de dos dimensiones si es bidimensional mediante las palabras clave **`blockIdx.x`** y **`blockIdx.y`**.Y los que se organizan en un espacio tridimensional como es nuestro caso general porque tenemos las coordenadas de los puntos en x,y,z, hemos definido los bloques con un máximo de 512 flujos de ejecución. La dimensión del flujo es accesible dentro del kernel a través de la siguiente variable **`blockDim.x y blockIdx.x`** como se muestra en ejemplo anterior de distance_to_vertices **`ip=threadIdx.x+blockDim.x*blockIdx.x.`** Cuando el procesador principal hace una llamada a un *kernel,* se tiene que especificar las dimensiones del grid y de los bloques de flujos mediante parámetros de configuración. El primer parámetro especifica las dimensiones del grid en términos de número de bloques y el segundo especifica las dimensiones de cada bloque en términos de número de flujo. En nuestro caso aparece como la siguiente variable dentro del fichero "point2trimeshCUDA" **`kern_vertices.ThreadBlockSize=512`** y **`kern_vertices.GridSize=[ceil(m/512),1].`**

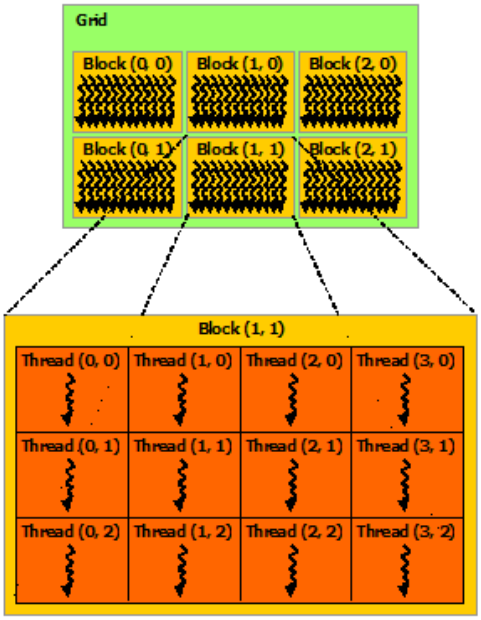


Figura 4.4.2. Grids por bloques de Threads

La ejecución típica de un programa CUDA se muestra en la Figura 4.4.2.1. La ejecución comienza en el procesador principal (CPU). Cuando se invoca un *kernel*, la ejecución se mueve hacia el dispositivo (GPU), donde se genera un número muy elevado de flujos (*grid*) como se ha descrito anteriormente. Un *Kernel* finaliza cuando todos sus flujos finalizan la ejecución en el *grid* correspondiente. Una vez finalizado el *kernel*, la ejecución del programa continúa en el procesador principal hasta que se invoca otro *kernel*.

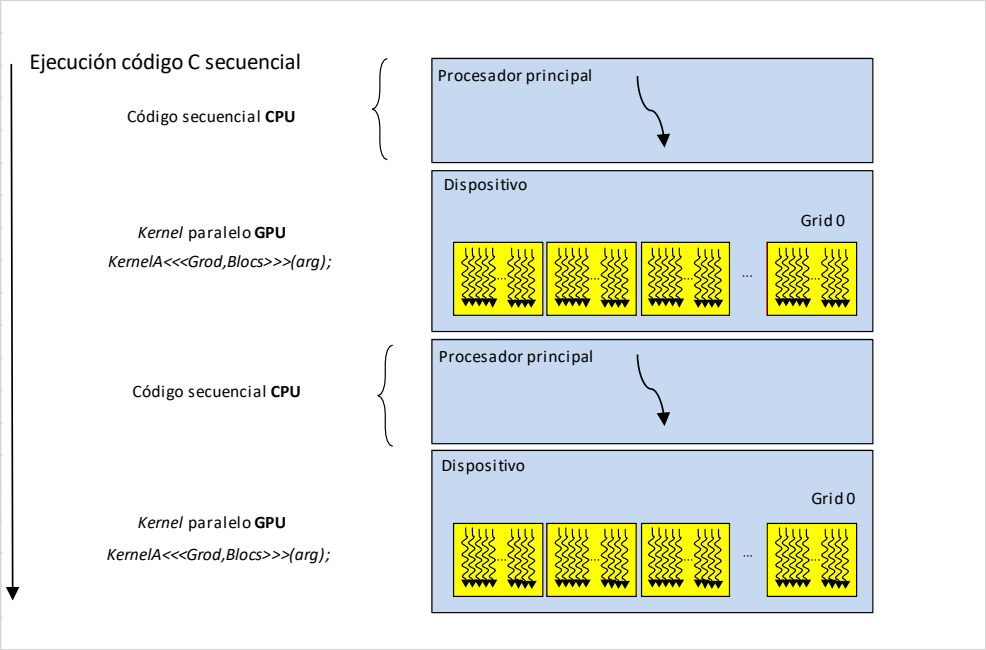


Figura 4.4.2.1 Etapas en la ejecución de un programa CUDA

4.4.3. Programación con CUDA

Y, por último, el tercer paso es hacer una versión en CUDA. MATLAB con parallel toolbox tiene un mecanismo para ejecutar Kernels CUDA. Los archivos resultantes del primer paso se transforman con cierta facilidad en Kernels CUDA como se ha explicado anteriormente, quitando el bucle, pero haciendo que cada "iteración" del bucle principal la haga en un *thread*. Cada *Kernel* CUDA se ejecutará a partir del procesador "point2trimeshCUDA" como se muestra en el listado 4.4.3, del cual cada uno corresponderá a cada función del algoritmo, 'kern_surface' que calcula las distancias de todos los puntos a todas las caras, 'kern_edges' calcula la distancia a todas las aristas y 'kern_vertices' calcula la distancia a todos los vértices.

Estas llamadas al *Kernel* desde el programa, consiste en la invocación del *kernel* desde un procesador principal, que corresponde al programa "point2trimesh" como se ha descrito anteriormente, escrito en C, donde se indica los parámetros necesarios para utilizar la GPU. Especificando las dimensiones del *grid*, de los bloques de flujos y tamaño de la memoria a utilizar. Además de los envíos de los datos de entrada y salida a la GPU con GPUArray. Cada *Kernel* se guarda con extensión .cu y en su llamada también introducimos la salida del fichero .ptx que corresponde al archivo ensamblador intermedio ejecutado tras el proceso de compilación.

En este programa como en cualquier otro programa CUDA tiene una o más fases que pueden ser ejecutadas o bien en el procesador principal (CPU) o bien en el dispositivo GPU. Las fases en las que hay muy poco o ningún paralelismo a nivel de datos se implementa en el código que se ejecutará en el procesador principal como es nuestro caso el fichero C, "point2trimeshCUDA", y las fases con un nivel de paralelismo a nivel de datos elevado se implementan en el código que se ejecutará en la GPU como son los *kernels.*

Listado 4.4.3: Procesador “point2trimeshCUDA”

```
…
function [D,P,F] = processPoint2(faces,vertices,qPoints,normals)
%[nqpoints,mq]=size(qPoints)----> nQPoints = size(qPoints,1);
[npoints,n]=size(qPoints);
[mrowsf,nfaces]=size(faces);
[mrowsv,nvertices]=size(vertices);
[mrowse,nedges]=size(edges);

edges = [faces(:,[1,2]); faces(:,[1,3]); faces(:,[2,3])]; % (#edges x 2) vertice IDs

d = gpuArray.NaN(npoints,3); % (distanceTypes x 1)
p = gpuArray.NaN(npoints,3,3); % (distanceTypes x xyz)
P= gpuArray.NaN(npoints,3);
f = gpuArray.NaN(npoints,3); % (distanceTypes x 1)
v=gpuArray.NaN(npoints,1);

kern_vertices=parallel.gpu.CUDAKernel('distance_vertices_CUDA.ptx','distance_vertice
s_CUDA.cu');
kern_edges=parallel.gpu.CUDAKernel('distance_edges_CUDA.ptx','distance_edges_CUDA.cu
');
kern_surfaces=parallel.gpu.CUDAKernel('distance_surfaces_CUDA.ptx','distance_surface
s_CUDA.cu');
kern_vertices.ThreadBlockSize=512;
kern_edges.ThreadBlockSize=512;
kern_surfaces.ThreadBlockSize=512;
kern_vertices.GridSize=[ceil(npoints/512),1];
kern_edges.GridSize=[ceil(npoints/512),1];
kern_surfaces.GridSize=[ceil(npoints/512),1];
…

```

4.4.4. Compilación del código en CUDA

El compilador de NVIDIA se encarga de proporcionar la parte del código del procesador principal y el dispositivo durante el proceso de compilación. En este caso el código del dispositivo se ha compilado con nvcc.

Normalmente la compilación de las aplicaciones CUDA se realizan con el *driver* compilador nvcc, este puede realizar muchas funciones, siguiendo caminos de compilación diferentes según las necesidades como se muestra en la Figura 4.4.4. Algunas funciones consisten en separar el código del anfitrión (*host*, en C) del dispositivo (*device*, con extensiones específicas CUDA C). Una vez separado los códigos lanza los compiladores específicos de cada tipo por separado, para después integrarlos en un único ejecutable como gcc.

En nuestro programa la compilación que más se asemeja de la siguiente imagen es la Figura de la derecha.

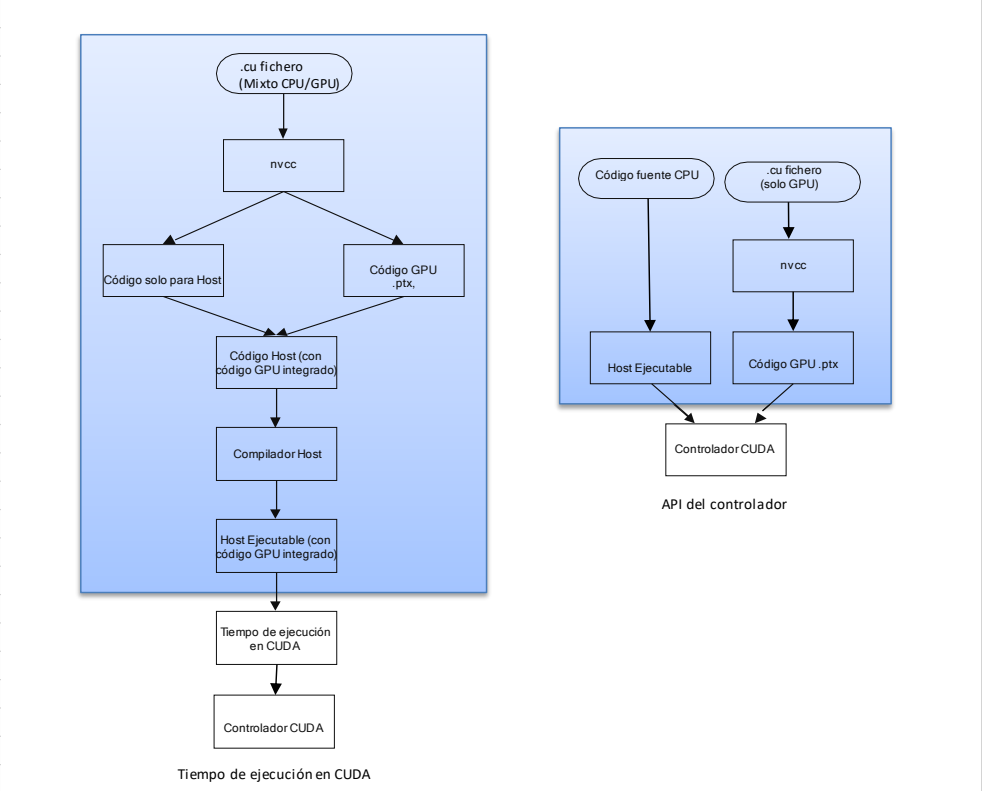


Figura 4.4.4 Diagramas de flujos de compilación en nvcc

La parte de compilación de la tarjeta gráfica es más específica, solo se puede realizar con una herramienta proporcionada por NVIDIA, como es el formato PTX (Parallel Thread eXecution), para el cual NVIDIA provee el compilador ptxas, este puede ser invocado directamente por nvcc.

La compilación del código se ha realizado con PTX como se muestra a continuación, es la representación intermedia del código a ejecutar en la GPU. Este formato es neutro, es decir, que es independiente de la arquitectura y de las capacidades computacionales de la tarjeta gráfica en la que finalmente se ejecutará. Puede ser compilado *offline* por el compilador driver nvcc, o compilado automáticamente *online* JIT según se necesite.

>> nvcc -ptx distance_vertices_CUDA.cu

La ventaja de compilar con los ptx es que son más portables, ya que pueden compilar a cualquier arquitectura o capacidad igual o superior a su formato. La compilación específica del ptx es que se tiene que realizar justo en el momento en que se va a lanzar el *kernel* CUDA, no durante la ejecución del inicio de la aplicación.

5. Utilidad del problema

En la actualidad, cualquier ingeniería utiliza softwares de alto rendimiento para la realización de cálculos masivos, estadisticas de rendimiento, proyecciones, estimaciones de tiempo de ejecucción, bases de datos, etc. Éstos softwares soportan mucha carga de almacenamiento, y en ocasiones produce una gran espera en la resolución del problema. Hoy en dia el tiempo es algo muy valorado en la eficacia del cálculo.

En este caso la toma masiva de datos, viene dada por un escaneo 3D realizado en la Universidad. Esta técnica es muy utilizada en varias ingenierias como puede ser la ingeniería industrial, ingeniería aeronautica, ingeniería civil, topografía, arquitectura y mineria. Una toma de datos con laser escaner puede llegar a capturar millones de puntos en muy poco tiempo. Estos datos son tratados en un software para analizar, editar y modelar lo que se ha capturado. Por lo que el cálculo del problema de este proyecto es muy utilizado en programas de modelados 3D, para la eliminación de ruido o elementos en movimiento,que son externos a la malla triangulada. Esta función también puede ser usada para calcular el grosor local de un objeto 3D y para comparar distintas tomas de nubes de puntos de la misma zona escaneada.

Uno de los software libre que utilizan esta función es CloudCompare que es un software de procesamiento de nubes de puntos 3D y mallas trianguladas. Este programa empezó como un proyecto de doctorado sobre detección de cambios en datos geometricos 3D. En ese momento el objetivo principal era detectar rápidamente los cambios en las nubes de puntos de alta densidad adquirida con laser escáner en instalaciones industriales o en construcción. Después evolucionó como software de procesamiento de datos 3D más general y avanzado de código abierto.

A lo que se quiere llegar es que esta función con un alto rendimiento puede ser muy valioso para algunas ingenierias.

6.Resultados

6.1 Casos de prueba

En este estudio se ha realizado varios casos de prueba para probar la eficacia del cálculo con las respectivas técnicas, comparando el tamaño de los datos de entrada, como se ha comentado anteriormente, la nube de puntos y la superficie triangulada. Se ha clasificado en tres categorías el caso sencillo que corresponde a los datos de la *Figura 2.1* con 3 puntos, el caso medio que corresponde a una matriz de caras de 23979x3, una matriz de vértices de 12171x3 y una matriz de 500 puntos, y el otro caso es el grande que corresponde a una matriz de caras de 12930x3, una matriz de vértices de 64912x3 y una matriz de 99917 puntos, pero en este caso también se ha querido comparar con distintos tamaños de la matriz de puntos con 50000, 25000 y 10000.

A continuación, se muestra la nube de puntos del escaneo donde se ha extraído tanto la matriz de vértices, la matriz de caras y la matriz de puntos. La Figura 6.1 corresponde al escaneo del caso medio, donde se ha recortado solo el maniquí para reducir tanto el número de puntos como de la malla triangulada, que se puede apreciar en la Figura 6.1.1.

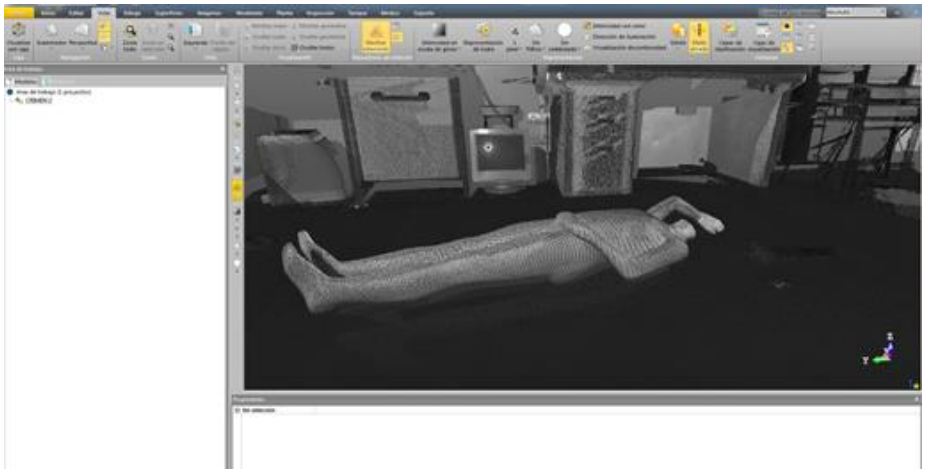


Figura 6.1: Escaneo 3D del maniquí

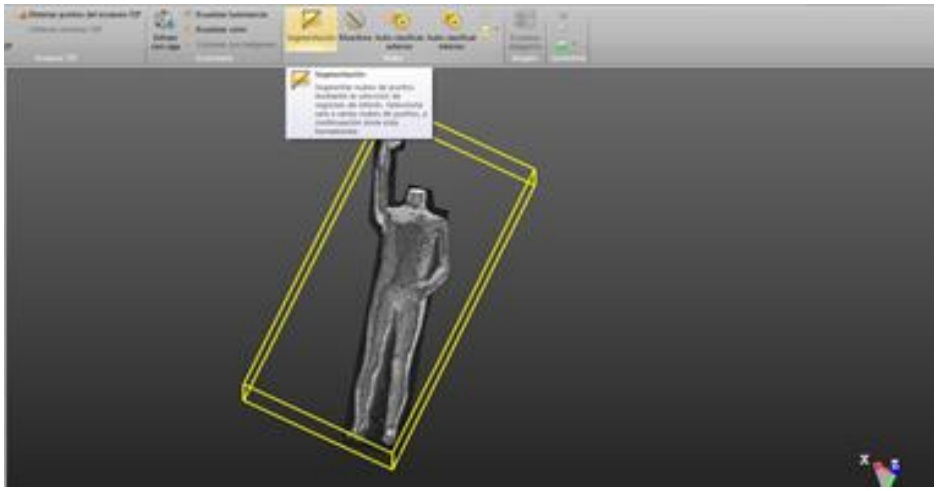


Figura 6.1.1: Ejemplo recorte nube de puntos

Y para el caso más grande se ha extraido la parte del escritorio que se visualiza en la siguiente imagen la Figura 6.1.2.

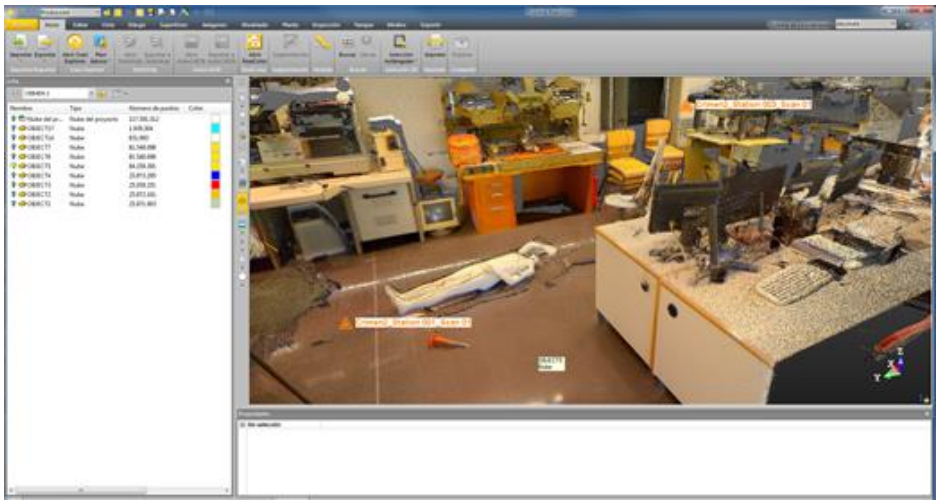


Figura 6.1.2: Escaneo 3D de todo el laboratorio

6.2 Resultados obtenidos

6.2.1 Resultados del caso medio

Directamente se va a mostrar los resultados obtenidos con el caso medio desde la computadora de Kempes de la universidad, ya que con el caso sencillo los resultados son insignificantes, debido a que el cálculo de las distancias se ha realizado solamente con tres puntos, la versión original tarda en ejecutarse 0.1033 segundos y las versiones paralelas los resultados obtenidos son inferiores a 0.0453 segundos. Además de obtener los tiempos de ejecución en MATLAB se ha utilizado el "plot3M" para comprobar desde una imagen que los resultados son correctos. Como se puede verificar en la Figura 6.2.1. los 500 puntos de este caso corresponden a los puntos rojos de la imagen que forman el maniquí de la Figura 6.1. y la malla triangulada en negro y gris.

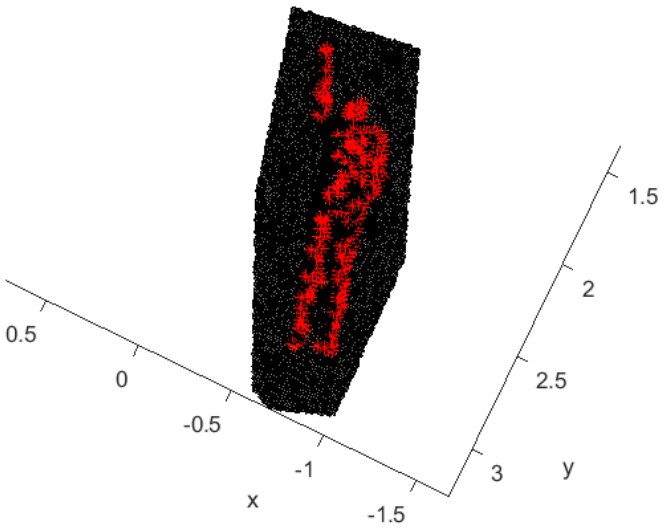


Figura 6.2.1: PLOT del caso medio en MATLAB

Los resultados que se observan a continuación se han comparado con la versión original de la función MATLAB "point2trimesh" con las distintas técnicas de paralelización utilizadas, como *parfor* el algoritmo paralelo de Parallel Toolbox (12 cores), la versión de bucles que corresponde a los ficheros MEX en un bucle en C, la versión GPU que corresponde a la paralelización en CUDA y por último con la versión OpenMP (24 cores). En esta Figura 6.2.1.1 se aprecia bien la diferencia del tiempo de ejecución de la versión original con todas las versiones paralelas, la versión original desde Kempes tarda 3.33 segundos en realizar el cálculo y el resto de versiones van en orden decreciente con resultados inferiores a 1 segundo de ejecución.

Resultados del caso medio en Kempes	Tiempo en segundos
Versión Original	3,33
Versión Original con Parallel	0,95
Versión MEX con Bucles	0,71
Versión GPU con CUDA	0,43
Versión MEX con OpenMP	0,11

Tabla 6.2.1: Resultados del tiempo de ejecución del caso medio en Kempes

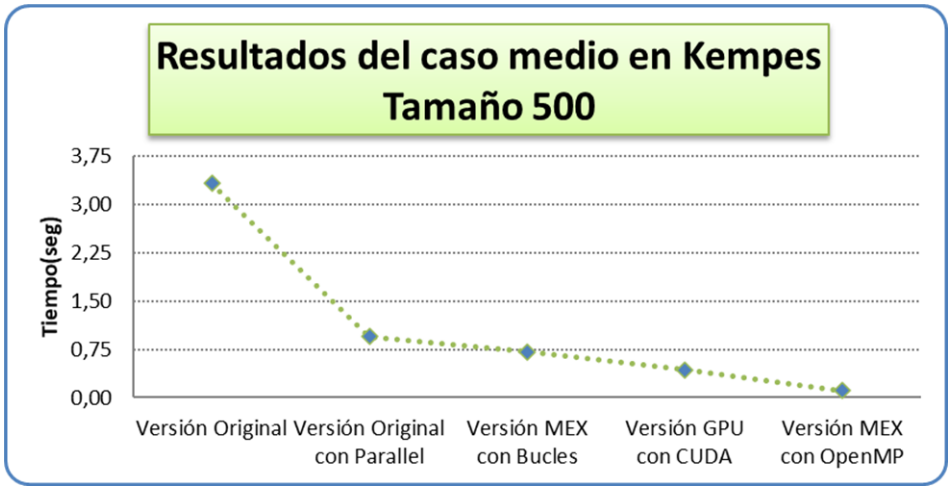


Figura 6.2.1.1: Resultados del caso medio en Kempes

Por lo que se observa en este caso, la versión más eficiente es la versión OpenMP con un resultado de 0,11 segundos de ejecución.

6.2.2 Resultados del caso grande

Como se ha mostrado en el apartado anterior la siguiente Figura 6.2.2 corresponde al resultado de la imagen de MATLAB, representando de la misma manera, pero con el caso grande. Los puntos rojos representan la matriz de 100000 puntos y la malla triangulada de fondo de negro y gris, la matriz de caras (123930x3) y vértices (64912x3).

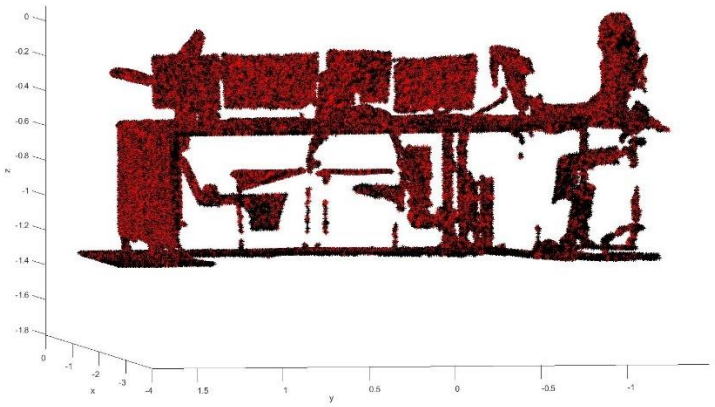


Figura 6.2.2: PLOT del caso grande en MATLAB

En este caso la versión original tarda demasiado, más de una hora. Comparada con las nuevas versiones incluso con la versión paralela con *parfor*, podemos ver en La Figura 6.2.2.1 que no es eficiente, hay mucha diferencia en tiempo de ejecución de 4150 segundos a 13,32 segundos con la versión GPU de CUDA. Observando el caso anterior, cuanto más grande son los archivos de entrada menos eficiente es la versión original del problema.

Resultados del caso grande en kempes	Tiempo en segundos
Versión Original	4150
Versión Original con Parallel	900
Versión MEX con Bucles	539
Versión GPU con CUDA	13,32
Versión MEX con OpenMP	40,87

Tabla 6.2.2: Resultados del tiempo de ejecución del caso grande en Kempes

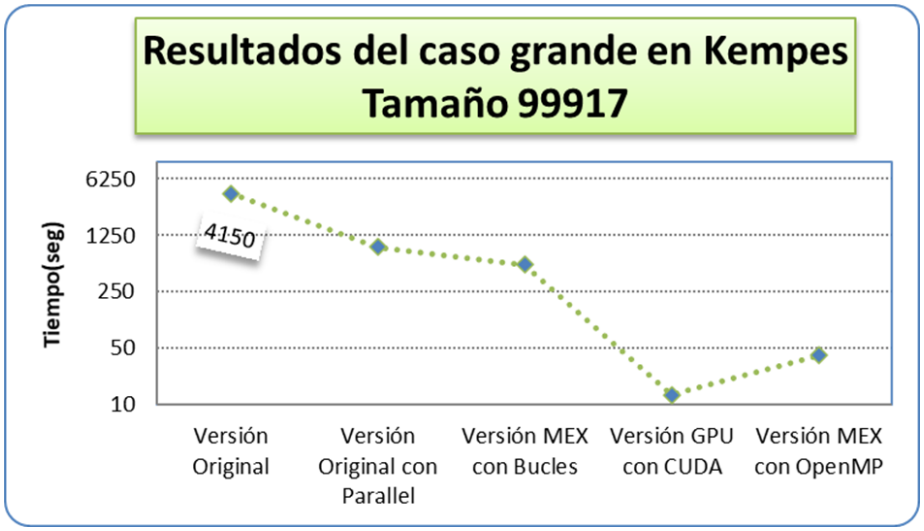


Figura 6.2.2.1: Resultados del caso grande en Kempes

En la siguiente gráfica 6.2.2.2. se aprecia la variación del tiempo de ejecución con el tamaño del problema usando el caso grande en Kempes y variando el número de puntos para 10000,25000 ,50000 y 100000 puntos. Las versiones paralelas toman en el mismo patrón cuanto menor es el tamaño de puntos menos tarda en ejecutarse, pero viendo la diferencia de las distintas versiones, tanto la versión con *parfor* y la versión de bucles, aunque el tiempo de ejecución es la mitad al otro sigue siendo resultados más grandes con respecto a la versión de GPU y OpenMP. Por lo que se podría decir en este caso la mejor opción para el cálculo del problema sería la versión de CUDA ya que sus resultados son los tiempos de menor ejecución.

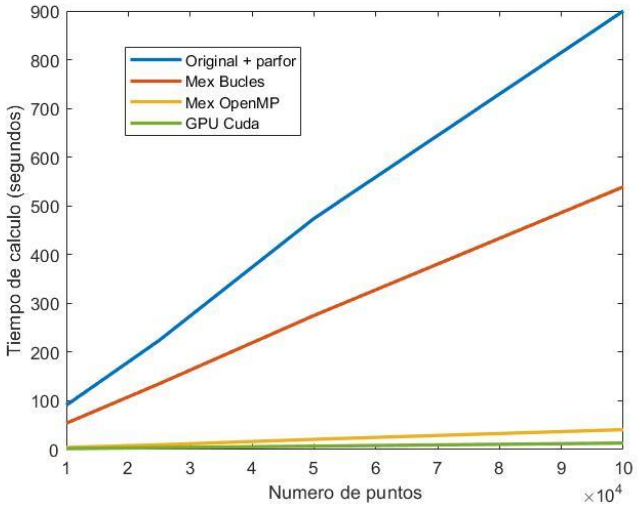


Figura 6.2.2.2: Análisis del caso grande con distintos tamaños de puntos

7.Conclusiones y trabajos futuros

En este proyecto como se ha visto en la introducción su función principal es ofrecer una mejora que permita acelerar operaciones de alto coste de almacenamiento. En nuestro caso se ha estudiado una función en concreto, que consiste en el cálculo de distancias entre una nube de puntos y una superficie triangulada en tres dimensiones.

Se ha reestructurado el código y desarrollado con versiones paralelas para la aceleración de este cálculo. Las versiones que se han desarrollado han sido la paralelización con *parfor* desde la versión original del código en MATLAB, paralelización mediante archivos MEX en C, paralelización mediante OpenMP y CUDA.

Todos los resultados han sido obtenidos desde la plataforma Kempes donde se ha comparado el tiempo de ejecución en la CPU desde MATLAB, la versión original del problema con respecto al tiempo de ejecución de las versiones nuevas, excepto la versión CUDA que su ejecución se realiza desde la GPU tesla K20Xm.

Analizando las prestaciones obtenidas en las evaluaciones de las distintas versiones. Todos los resultados han sido inferiores al tiempo de ejecución desde la versión original. La primera versión paralela con *parfor*, reduce el tiempo de ejecución con 78%. La versión con código C compilado secuencial es un 40% más rápida que la versión original MATLAB usando *parfor* (con 12 cores). Cuando se compara la versión C con OpenMP (24 cores) con la versión original de MATLAB con *parfor* (12 cores) la primera es 92 % más rápida que la segunda. Pero donde nos es más importante es la comparación de resultados entre las dos últimas versiones OpenMP y CUDA que se puede apreciar en la Figura 6.2.2.2. Aunque las memorias son compartidas la ejecución del cálculo se realiza en distintos procesadores, uno en la unidad central de procesamiento (CPU) con 24 Cores y otro en la unidad de procesamiento gráfico (GPU) con 2688 Cores. La versión CUDA es la que mejor resultado ofrece en tiempo de ejecución, con 13.32 segundos y la versión OpenMP con 40.87 segundos, como se muestra en la Tabla 6.2.2. Por lo que podemos llegar a la conclusión con esta evaluación de resultados es que el desarrollo de la función de “Point2trimesh” con CUDA en GPUs es la más eficaz para el cálculo de distancias entre una nube de puntos y una superficie triangulada en tres dimensiones.

Por último, el posible trabajo futuro que se podría realizar es el siguiente;

Optimizar el código implementado en CUDA sobre las nuevas tarjetas gráficas GeForce RTX 40 que llegarán en el tercer trimestre de este año. Esta nueva GPU AD102 [10] puede tener hasta 18432 núcleos CUDA. Tras a ver visto los resultados obtenidos desde el código implementado en CUDA al añadir más núcleos, el cálculo se realizará en más hilos por lo que el resultado será más optimo.

8.Apendice

Distance_to_vertices_CUDA.cu

```
void __global__ distance_vertices_CUDA(const double *qPoint,
                                       const double *faces,
                                       const double *vertices,
                                       const double *normals,
                                       int npoints,
                                       int mrowsv,
                                       int mrowsf,
                                       double *D,
                                       double *P,
                                       double *F,
                                       double *V
                                       )
{
 int ip=threadIdx.x+blockDim.x*blockIdx.x;

     if (ip<npoints)

{
          double mind,coefficients,coef0,coef1,coef2,auxd;
          int find,sgn,coeflargest ,i,j, imind;

          mind=100000;
          imind=0;


     for (i=0; i<mrowsv; i++)
      {
          auxd=0;

          for(j=0; j<3; j++){
          auxd = auxd + pow(vertices[i+j*mrowsv]- qPoint[ip +j*npoints],2);}

          if (auxd< mind)
             {
              mind=auxd;
              imind=i;
               }
  }


          D[ip] = sqrt(mind);
          V[ip] = imind +1;
          P[ip]= vertices[imind];
          P[ip+npoints]= vertices[imind+1*mrowsv];
          P[ip+2*npoints]= vertices[imind+2*mrowsv];

          /**/
```

distance_to_vertices_CUDA.cu

```
48            coeflargest=0;
49
50
51        for (i=0; i<mrowsf; i++)
52
53         {
54
55            for(j=0; j<3; j++){
56
57            if (faces[i+j*mrowsf]==imind){
58
59
60                if ( find==0) {
61                    {F[ip] = i + 1;
62                    find = 1;
63                }
64
65                coef0=0;
66                coef1=0;
67                coef2=0;
68                coef0+=normals[i]*(qPoint[ip]- P[ip]);
69                coef1+=normals[i+1*mrowsf]*(qPoint[ip+npoints]- P[ip+npoints]);
70                coef2+=normals[i+2*mrowsf]*(qPoint[ip+2*npoints]- P[ip+2*npoints]);
72                coefficients=coef0+coef1+coef2;
73                }
74
75            if (abs(coefficients)> abs(coeflargest))
76               {
77                coeflargest=coefficients;
78
79               }
80        }
81     }
82            if ( coeflargest>= 0){
83            sgn=1;}
84            else {
85            sgn=-1;}
86
87          D[ip]=(D[ip])*sgn;
88    }
89    }
90    }
91
```

distance_to_edges_mex.c

```c
#include <math.h>
#include <stdio.h>
#include "mex.h"

void distance_to_edges_mex(double *vertices,double *faces, double *qPoint, double
*normals, double *r1,double *r2,double *edges,double *D, double *P,double
*F,mwSize mrowse,mwSize mrowsf )
{
 mwSize ip;
 for (ip=0; ip<npoints; ip++) {
      mwSize i,I;
      double num,den,t,mind,coefficients,Paux[3],daux;
      double aux0,aux1,aux2,rest1,rest0,rest2,P0,P1,P2,comp1,comp2,comp0,D0,D1,D2,
      n0,n1,n2,coef0,coef1,coef2;
      int find,sgn,coeflargest,cond1,cond2;

      t=0;
      num=0;
      den=0;
      I=0;
      mind=mxGetInf();


      for (i=0; i<mrowse; i++)
      {

         /*%t = dot( bsxfun(@minus,qPoint,r1), r2-r1, 2) ./ sum((r2-r1).^2,2); %
(#edges x 1) location of intersection relative to r1 and r2*/

             aux0=0, rest0=0;
             aux1=0, rest1=0;
             aux2=0, rest2=0;

             rest0=r2[i]-r1[i];
             rest1=r2[i+mrowse]-r1[i+mrowse];
             rest2=r2[i+2*mrowse]-r1[i+2*mrowse];

             aux0+=rest0*(qPoint[ip]- r1[i]);
             aux1+=rest1*(qPoint[ip+npoints]- r1[i+mrowse]);
             aux2+=rest2*(qPoint[ip+2*npoints]- r1[i+2*mrowse]);

             num=aux0+aux1+aux2;

             den=(rest0*rest0)+(rest1*rest1)+(rest2*rest2);

             t= num/den;
```

```
49  /*% Distance between intersection and query point*/
50        if (t>0){
51             if(t<1)
52             {
53              P0=0; comp0=0; D0=0;
54              P1=0; comp1=0; D1=0;
55              P2=0; comp2=0; D2=0;
56
57              comp0=(r2[i]-r1[i])*t;
58              comp1=(r2[i+mrowse]-r1[i+mrowse])*t;
59              comp2=(r2[i+2*mrowse]-r1[i+2*mrowse])*t;
60
61              P0=r1[i]+comp0;
62              P1=r1[i+mrowse]+comp1;
63              P2=r1[i+2*mrowse]+comp2;
64
65              Paux[0]=P0;
66              Paux[1]=P1;
67              Paux[2]=P2;
68
69              D0=qPoint[ip]-P0;
70              D1=qPoint[ip+npoints]-P1;
72              D2=qPoint[ip+2*npoints]-P2;
73
74              daux=sqrt((D0*D0)+(D1*D1)+(D2*D2)); //mod4 daux
75
76               if (daux< mind)
77             {
78              mind=daux;
79              D[ip]=daux;
80              I=ip;
81              P[ip]=Paux[0];
82              P[ip+npoints]=Paux[1];
83              P[ip+2*npoints]=Paux[2];    }
84             }
85         }
86     }
87
88     find=0; //mod3
89     for (i=0; i<mrowsf; i++)
90      {
91     /*% find faces that belong to the edge*/
92
93      cond1=(edges[I]==faces[i])||(edges[I]==faces[i+mrowsf])||(edges[I]==faces[i+2*
94      mrowsf]);
95
96      cond2=(edges[mrowse+I]==faces[i])||(edges[mrowse+I]==faces[i+mrowsf])||(edges
97      [mrowse+I]==faces[i+2*mrowsf]);
98
99        if (cond1==1&&cond2==1){
100
```

```
101       if (cond1==1&&cond2==1){
102           if (find==0)
103               {F[ip]=i+1;
104                find=1;
105               }
106               n0=normals[i];
107               n1=normals[i+mrowsf];
108               n2=normals[i+2*mrowsf];
109               /*% scalar product between distance vector and normal vectors*/
110                 coef0=0;
111                 coef1=0;
112                 coef2=0;
113                 coef0+=n0*(qPoint[ip]- P[ip]);
114                 coef1+=n1*(qPoint[ip+npoints]- P[ip+npoints]);
115                 coef2+=n2*(qPoint[ip+2*npoints]- P[ip+2*npoints]);
116                 coefficients=coef0+coef1+coef2;
117
118        if (abs(coefficients)> abs(coeflargest)
119               {
120                 coeflargest=coefficients;
121                 }
122       }
123 }
124 if ( coeflargest>= 0){
125         sgn=1;}
126         else {
127         sgn=-1;}
128
129        (D[ip])=(D[ip])*sgn;
130 }
131
132 void mexFunction( int nlhs, mxArray *plhs[],
133                     int nrhs, const mxArray *prhs[])
134 {
135       double *vertices, *qPoint, *faces, *normals,*edges,*r1,*r2, *D, *P, *F;
136       size_t mrowse,mrowsf;
137       /* check for proper number of arguments */
138       if(nrhs!=6) {
139           mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs","seven inputs required.");
140       }
141       if(nlhs!=3) {
142           mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs","three output required.");
143       }
144
145       /*  get dimensions of the input matrix rows  */
146
147       mrowse=mxGetM(prhs[5]);
148       mrowsf=mxGetM(prhs[0]);
149       npoints=mxGetM(prhs[1]);
150
```

distance_to_edges_mex.c

```
151         /* get dimensions of the input matrix cols*/
152         /*ncols = mxGetN(prhs[0]);
153 
154         /* create a pointer to the real data in the input   */
155 
156         faces=mxGetPr(prhs[0]);
157         //vertices=mxGetPr(prhs[1]);
158         qPoint=mxGetPr(prhs[1]);
159         normals=mxGetPr(prhs[2]);
160         r1=mxGetPr(prhs[3]);
161         r2=mxGetPr(prhs[4]);
162         edges=mxGetPr(prhs[5]);
163 
164         /* create the output matrix */
165         plhs[0] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)1,mxREAL);
166         plhs[1] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)3,mxREAL);
167         plhs[2] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)1,mxREAL);
168 
169 
170         /* get a pointer to the real data in the output  */
171          D= mxGetPr(plhs[0]);
172          P= mxGetPr(plhs[1]);
173          F= mxGetPr(plhs[2]);
174 
175 
176         /* call the computational routine */
177 
178    distance_to_edges_mex(vertices,faces,qPoint,normals,r1,r2,edges,D,P,F,mrowse,mrowsf
179    ,npoints);
180 }
181 
```

distance_to_edges_omp.c

```c
#include <math.h>
#include <stdio.h>
#include "mex.h"



void distance_to_edges_omp(double *faces, double *qPoint, double *normals, double
*r1,double *r2,double *edges,double *D, double *P,double *F,mwSize mrowse,mwSize
mrowsf ,mwSize npoints)
{
 mwSize ip;
 #pragma omp parallel {
 #pragma omp for

 for (ip=0; ip<npoints; ip++) {

    mwSize i, I;
    double num,den,t,mind,coefficients,Paux[3],daux;
    double aux0,aux1,aux2,rest1,rest0,rest2,P0,P1,P2,comp1,comp2,comp0,D0,D1,D2,
    n0,n1,n2,coef0,coef1,coef2;
    int find,sgn,coeflargest,cond1,cond2;

    t=0;
    num=0;
    den=0;
    I=0;
    //minD=10000000;
    mind=mxGetInf();


    for (i=0; i<mrowse; i++)
      {

        /*%t = dot( bsxfun(@minus,qPoint,r1), r2-r1, 2) ./ sum((r2-r1).^2,2); %
(#edges x 1) location of intersection relative to r1 and r2*/

            aux0=0, rest0=0;
            aux1=0, rest1=0;
            aux2=0, rest2=0;

            rest0=r2[i]-r1[i];
            rest1=r2[i+mrowse]-r1[i+mrowse];
            rest2=r2[i+2*mrowse]-r1[i+2*mrowse];

            aux0+=rest0*(qPoint[ip]- r1[i]);
            aux1+=rest1*(qPoint[ip+npoints]- r1[i+mrowse]);
            aux2+=rest2*(qPoint[ip+2*npoints]- r1[i+2*mrowse]);
            num=aux0+aux1+aux2;

            den=(rest0*rest0)+(rest1*rest1)+(rest2*rest2);

            t= num/den;
```

distance_to_edges_omp.c

```
54  /*% Distance between intersection and query point*/
55          if (t>0){
56               if(t<1)
57               {
58                P0=0; comp0=0; D0=0;
59                P1=0; comp1=0; D1=0;
60                P2=0; comp2=0; D2=0;
61
62                comp0=(r2[i]-r1[i])*t;
63                comp1=(r2[i+mrowse]-r1[i+mrowse])*t;
64                comp2=(r2[i+2*mrowse]-r1[i+2*mrowse])*t;
65
66                P0=r1[i]+comp0;
67                P1=r1[i+mrowse]+comp1;
68                P2=r1[i+2*mrowse]+comp2;
69
70                Paux[0]=P0;
72                Paux[1]=P1;
73                Paux[2]=P2;
74
75                D0=qPoint[ip]-P0;
76                D1=qPoint[ip+npoints]-P1;
77                D2=qPoint[ip+2*npoints]-P2;
78
79                daux=sqrt((D0*D0)+(D1*D1)+(D2*D2));
80
81                 if (daux< mind)
82               {
83                mind=daux;
84                D[ip]=daux;
85                I=ip;
86                P[ip]=Paux[0];
87                P[ip+npoints]=Paux[1];
88                P[ip+2*npoints]=Paux[2];
89
90                }
91               }
92          }
93      }
94
95      find=0;
96      for (i=0; i<mrowsf; i++)
97       {
98      /*% find faces that belong to the edge*/
99
100     cond1=(edges[I]==faces[i])||(edges[I]==faces[i+mrowsf])||(edges[I]==faces[i+2*
101     mrowsf]);
102
103     cond2=(edges[mrowse+I]==faces[i])||(edges[mrowse+I]==faces[i+mrowsf])||(edges
104     [mrowse+I]==faces[i+2*mrowsf]);
105
```

```
106       if (cond1==1&&cond2==1){
107           if (find==0)
108               {F[ip]=i+1;
109                find=1;
110              }
111              n0=normals[i];
112              n1=normals[i+mrowsf];
113              n2=normals[i+2*mrowsf];
114              /*% scalar product between distance vector and normal vectors*/
115                coef0=0;
116                coef1=0;
117                coef2=0;
118                coef0+=n0*(qPoint[ip]- P[ip]);
119                coef1+=n1*(qPoint[ip+npoints]- P[ip+npoints]);
120                coef2+=n2*(qPoint[ip+2*npoints]- P[ip+2*npoints]);
121                coefficients=coef0+coef1+coef2;
122
123        if (abs(coefficients)> abs(coeflargest)
124              {
125                coeflargest=coefficients;
126                }
127       }
128 }
129 if ( coeflargest>= 0){
130          sgn=1;}
131          else {
132          sgn=-1;}
133
134         (D[ip])=(D[ip])*sgn;
135 }
136 }
137
138 void mexFunction( int nlhs, mxArray *plhs[],
139                     int nrhs, const mxArray *prhs[])
140 {
141       double *vertices, *qPoint, *faces, *normals,*edges,*r1,*r2, *D, *P, *F;
142       size_t mrowse,mrowsf, npoints;
143       /* check for proper number of arguments */
144       if(nrhs!=6) {
145           mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs","seven inputs required.");
146       }
147       if(nlhs!=3) {
148           mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs","three output required.");
149       }
150
151       /*  get dimensions of the input matrix rows  */
152
153       mrowse=mxGetM(prhs[3]);
154       mrowsf=mxGetM(prhs[0]);
155       npoints=mxGetM(prhs[1]);
```

distance_to_edges_omp.c

```
156         /* get dimensions of the input matrix cols*/
157         /*ncols = mxGetN(prhs[0]);
158 
159         /* create a pointer to the real data in the input   */
160 
161         faces=mxGetPr(prhs[0]);
162         //vertices=mxGetPr(prhs[1]);
163         qPoint=mxGetPr(prhs[1]);
164         normals=mxGetPr(prhs[2]);
165         r1=mxGetPr(prhs[3]);
166         r2=mxGetPr(prhs[4]);
167         edges=mxGetPr(prhs[5]);
168 
169         /* create the output matrix */
170         plhs[0] = mxCreateDoubleMatrix((mwSize) npoints,(mwSize)1,mxREAL);
171         plhs[1] = mxCreateDoubleMatrix((mwSize) npoints,(mwSize)3,mxREAL);
172         plhs[2] = mxCreateDoubleMatrix((mwSize) npoints,(mwSize)1,mxREAL);
173 
174 
175         /* get a pointer to the real data in the output  */
176          D= mxGetPr(plhs[0]);
177          P= mxGetPr(plhs[1]);
178          F= mxGetPr(plhs[2]);
179 
180 
181         /* call the computational routine */
182         distance_to_edges_omp(faces,qPoint,normals,r1,r2,edges,D,P,F,mrowse,mrowsf,
183     npoints);
184 }
185 
```

distance_to_edges_CUDA.cu

```
  void __global__ distance_edges_CUDA(const double *qPoint,
                                      const double *faces,
                                      const double *normals,
                                      const double *edges,
                                      double *r1,
                                      double *r2,
                                      double *D,
                                      double *P,
                                      double *F,
                                      int npoints,
                                      int mrowse,
                                      int mrowsf
                                     )
{
 int ip=threadIdx.x+blockDim.x*blockIdx.x;

     if (ip<npoints)

{

     double num,den,t,mind,coefficients,Paux[3],daux;
     double aux0,aux1,aux2,rest1,rest0,rest2,P0,P1,P2,comp1,comp2,
     comp0,D0,D1,D2,n0,n1,n2,coef0,coef1,coef2;
     int find,sgn,coeflargest,cond1,cond2,i,I;

     t=0;
     num=0;
     den=0;
     I=0;
     mind=100000;



     for (i=0; i<mrowse; i++)
       {

         /*%t = dot( bsxfun(@minus,qPoint,r1), r2-r1, 2) ./ sum((r2-r1).^2,2); %
(#edges x 1) location of intersection relative to r1 and r2*/

             aux0=0, rest0=0;
             aux1=0, rest1=0;
             aux2=0, rest2=0;

             rest0=r2[i]-r1[i];
             rest1=r2[i+mrowse]-r1[i+mrowse];
             rest2=r2[i+2*mrowse]-r1[i+2*mrowse];

             aux0+=rest0*(qPoint[ip]- r1[i]);
             aux1+=rest1*(qPoint[ip+npoints]- r1[i+mrowse]);
             aux2+=rest2*(qPoint[ip+2*npoints]- r1[i+2*mrowse]);

```

```
53                num=aux0+aux1+aux2;
54                den=(rest0*rest0)+(rest1*rest1)+(rest2*rest2);
55
56                t= num/den;
57
58              /*% Distance between intersection and query point*/
59           if (t>0){
60                if(t<1)
61                {
62                 P0=0; comp0=0; D0=0;
63                 P1=0; comp1=0; D1=0;
64                 P2=0; comp2=0; D2=0;
65
66                 comp0=(r2[i]-r1[i])*t;
67                 comp1=(r2[i+mrowse]-r1[i+mrowse])*t;
68                 comp2=(r2[i+2*mrowse]-r1[i+2*mrowse])*t;
69
70                 P0=r1[i]+comp0;
72                 P1=r1[i+mrowse]+comp1;
73                 P2=r1[i+2*mrowse]+comp2;
74
75                Paux[0]=P0;
76                Paux[1]=P1;
77                Paux[2]=P2;
78
79                 D0=qPoint[ip]-P0;
80                 D1=qPoint[ip+npoints]-P1;
81                 D2=qPoint[ip+2*npoints]-P2;
82
83                 daux=sqrt((D0*D0)+(D1*D1)+(D2*D2));
84
85                  if (daux< mind)
86                {
87                 mind=daux;
88                 D[ip]=daux;
89                 I=i;
90                 P[ip]=Paux[0];
91                 P[ip+npoints]=Paux[1];
92                 P[ip+2*npoints]=Paux[2];
93                 }
94                }
95            }
96         }
97         find=0;
98         for (i=0; i<mrowsf; i++)
99           {
100        /*% find faces that belong to the edge*/
101
102           cond1=(edges[I]==faces[i])||(edges[I]==faces[i+mrowsf])||(edges[I]==faces[i
103           +2*mrowsf]);
104
105           cond2=(edges[mrowse+I]==faces[i])||(edges[mrowse+I]==faces[i+mrowsf])||(edg
              es[mrowse+I]==faces[i+2*mrowsf]);
```

distance_to_edges_CUDA.cu

```
106
107          if (cond1==1&&cond2==1){
108
109              if (find==0)
110                  {F[ip]=i+1;
111                  find=1;
112              }
113              n0=normals[i];
114              n1=normals[i+mrowsf];
115              n2=normals[i+2*mrowsf];
116              /*% scalar product between distance vector and normal vectors*/
117                coef0=0;
118                coef1=0;
119                coef2=0;
120                coef0+=n0*(qPoint[ip]- P[ip]);
121                coef1+=n1*(qPoint[ip+npoints]- P[ip+npoints]);
122                coef2+=n2*(qPoint[ip+2*npoints]- P[ip+2*npoints]);
123                coefficients=coef0+coef1+coef2;
124
125          if (abs(coefficients)> abs(coeflargest))
126                {
127                 coeflargest=coefficients;
128                 }
129
130      }
131
132  }
133   if ( coeflargest>= 0){
134          sgn=1;}
135          else {
136          sgn=-1;}
137
138          D[ip]=(D[ip])*sgn;
139  }
140  }
141
```

distance_to_surfaces_mex.c

```
#include <math.h>
#include <stdio.h>
#include "mex.h"

void distance_to_surfaces_mex(double *faces, double *vertices,double *qPoint,
double *normals, double *r1,double *r2, double *r3,double *D, double *P,double
*F, mwSize mrowsf ,mwSize npoints)
{   mwSize ip;
    for (ip=0; ip<npoints; ip++)
      {
       mwSize i,I;
       double vq[3],rD[3],Paux[3],Daux,r31r31,r21r21,r21r31,r31vq,r21vq,d,
       bary[3],mind;
       double vq1,vq2,vq0,aux0,aux1,aux2,rD0,rD1,rD2,P0,P1,P2,rest0_31,rest1_31,
       rest2_31,rest0_21,rest1_21,rest2_21,aux0_r21r31,aux1_r21r31,aux2_r21r31,
       aux2_r31vq,aux1_r31vq,aux0_r31vq,aux0_r21vq,aux1_r21vq,aux2_r21vq;
       I=0;
       mind=mxGetInf();

      for (i=0; i<mrowsf; i++)
        { /*vq = bsxfun(@minus,point,r1);  % (#faces x 3)*/

          vq0=qPoint[ip]- r1[i];
          vq1=qPoint[ip+npoints]- r1[i+mrowsf];
          vq2=qPoint[ip+2*npoints]- r1[i+2*mrowsf];

          vq[0]=vq0;
          vq[1]=vq1;
          vq[2]=vq2;

          /*%D = dot(vq,normals,2); % (#faces x 1) distance to surface*/

          aux0+=vq[0]*normals[i];
          aux1+=vq[1]*normals[i+mrowsf];
          aux2+=vq[2]*normals[i+2*mrowsf];
          Daux=aux0+aux1+aux2;

          /*rD = bsxfun(@times,normals,D); % (#faces x 3) vector from surface to
query point*/

          rD0=normals[i]*Daux;
          rD1=normals[i+mrowsf]*Daux;
          rD2=normals[i+2*mrowsf]*Daux;

          rD[0]=rD0;
          rD[1]=rD1;
          rD[2]=rD2;

          /*P = bsxfun(@minus,point,rD);*/

          P0=qPoint[ip]- rD[0];
          P1=qPoint[ip+npoints]- rD[1];
          P2=qPoint[ip+2*npoints]- rD[2];
```

```
54            Paux[0]=P0;
55            Paux[1]=P1;
56            Paux[2]=P2;
57
58            /*find barycentric coordinates (query point as linear combination of two
59    edges)*/
60
61            rest0_31=r3[i]-r1[i];
62            rest1_31=r3[i+mrowsf]-r1[i+mrowsf];
63            rest2_31=r3[i+2*mrowsf]-r1[i+2*mrowsf];
64
65            r31r31=(rest0_31*rest0_31)+(rest1_31*rest1_31)+(rest2_31*rest2_31);
66
67            /*.....................................................................*/
68
69            rest0_21=r2[i]-r1[i];
70            rest1_21=r2[i+mrowsf]-r1[i+mrowsf];
72            rest2_21=r2[i+2*mrowsf]-r1[i+2*mrowsf];
73
74            r21r21=(rest0_21*rest0_21)+(rest1_21*rest1_21)+(rest2_21*rest2_21);
75
76            /*.....................................................................*/
77
78            aux0_r21r31=rest0_21*rest0_31;
79            aux1_r21r31=rest1_21*rest1_31;
80            aux2_r21r31=rest2_21*rest2_31;
81
82            r21r31=aux0_r21r31 + aux1_r21r31 + aux2_r21r31;
83
84            /*.....................................................................*/
85
86            aux0_r31vq=rest0_31*vq[0];
87            aux1_r31vq=rest1_31*vq[1];
88            aux2_r31vq=rest2_31*vq[2];
89
90            r31vq=aux0_r31vq + aux1_r31vq + aux2_r31vq;
91
92            /*.....................................................................*/
93
94            aux0_r21vq=rest0_21*vq[0];
95            aux1_r21vq=rest1_21*vq[1];
96            aux2_r21vq=rest2_21*vq[2];
97
98            r21vq=aux0_r21vq + aux1_r21vq + aux2_r21vq;
99
100           /*.....................................................................*/
101
102           d=(r31r31*r21r21)-(r21r31*r21r31);
103           bary[0]=((r21r21*r31vq)-(r21r31*r21vq))/d;
104           bary[1]=((r31r31*r21vq)-(r21r31*r31vq))/d;
105           bary[2]= 1- bary[0]-bary[1];
106
```

```
106                 if  ( ( abs(d)>1e-15) && (bary[0]>0) && (bary[1]>0) && (bary[2]>0)
107             && (bary[0]<1) && (bary[1]<1) && (bary[2]<1)){
108
109     /*% find nearest face for query point*/
110                 if (Daux< mind)
111                {
112                 mind=Daux;
113                 I=ip;
114
115                 P[ip]=Paux[ip];
116                 P[ip+npoints]=Paux[ip+npoints];
117                 P[ip+2*npoints]=Paux[ip+2*npoints];
118                 F[ip]=ip+1;
119                 }
120              }
121                 else if(ip==0)
122                   {F[ip]=1;
123                    P[ip]=Paux[ip];
124                    P[ip+npoints]=Paux[ip+npoints];
125                    P[ip+2*npoints]=Paux[ip+2*npoints];
126               }
127         }
128          D[ip]=mind;}
129     }
130     void mexFunction( int nlhs, mxArray *plhs[],
131                       int nrhs, const mxArray *prhs[])
132     {
133         double *vertices, *point, *faces, *normals,*r3,*r1,*r2, *D, *P, *F;
134         size_t mrowsf,npoints;
135         /* check for proper number of arguments */
136         if(nrhs!=6) {
137             mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs","seven inputs required.");
138         }
139         if(nlhs!=3) {
140             mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs","three output required.");
141         }
142
143         /*  get dimensions of the input matrix rows  */
144
145         mrowsf=mxGetM(prhs[0]);
146         npoints=mxGetM(prhs[1]);
147         /* get dimensions of the input matrix cols*/
148         /*ncols = mxGetN(prhs[0]);
149
150         /* create a pointer to the real data in the input   */
151
152         faces=mxGetPr(prhs[0]);
153         //vertices=mxGetPr(prhs[1]);
154         point=mxGetPr(prhs[1]);
155         normals=mxGetPr(prhs[2]);
156         r1=mxGetPr(prhs[3]);
157         r2=mxGetPr(prhs[4]);
158         r3=mxGetPr(prhs[5]);
159
```

distance_to_surfaces_mex.c

```
        /* create the output matrix */
      plhs[0] = mxCreateDoubleMatrix((mwSize)1,(mwSize)1,mxREAL);
      plhs[1] = mxCreateDoubleMatrix((mwSize)1,(mwSize)3,mxREAL);
      plhs[2] = mxCreateDoubleMatrix((mwSize)1,(mwSize)1,mxREAL);

        /* get a pointer to the real data in the output  */
       D= mxGetPr(plhs[0]);
       P= mxGetPr(plhs[1]);
       F= mxGetPr(plhs[2]);

       /* call the computational routine */

distance_to_surfaces_mex(faces,vertices,point,normals,r1,r2,r3,D,P,F,mrowsf,npoi
nts);
```

distance_to_surfaces_omp.c

```
#include <math.h>
#include <stdio.h>
#include "mex.h"


void distance_to_surfaces_omp(double *faces,double *qPoint, double *normals,
double *r1,double *r2, double *r3,double *D, double *P,double *F, mwSize mrowsf,
mwSize npoints )
{
    mwSize ip;

    #pragma omp parallel {
    #pragma omp for

    for (ip=0; ip<npoints; ip++)
      {
         mwSize i,I;
         double vq[3],rD[3],Paux[3],Daux,r31r31,r21r21,r21r31,r31vq,r21vq,d,
         bary[3],mind;
         double vq1,vq2,vq0,aux0,aux1,aux2,rD0,rD1,rD2,P0,P1,P2,rest0_31,rest1_31,
         rest2_31,rest0_21,rest1_21,rest2_21,aux0_r21r31,aux1_r21r31,aux2_r21r31,
         aux2_r31vq,aux1_r31vq,aux0_r31vq,aux0_r21vq,aux1_r21vq,aux2_r21vq;
         I=0;
         mind=mxGetInf();

         for (i=0; i<mrowsf; i++)
           { /*vq = bsxfun(@minus,point,r1);  % (#faces x 3)*/

              vq0=qPoint[ip]- r1[i];
              vq1=qPoint[ip+npoints]- r1[i+mrowsf];
              vq2=qPoint[ip+2*npoints]- r1[i+2*mrowsf];

              vq[0]=vq0;
              vq[1]=vq1;
              vq[2]=vq2;

          /*%D = dot(vq,normals,2); % (#faces x 1) distance to surface*/
```

```
38              aux0+=vq[0]*normals[i];
39              aux1+=vq[1]*normals[i+mrowsf];
40              aux2+=vq[2]*normals[i+2*mrowsf];
41
42              Daux=aux0+aux1+aux2;
43
44             /*rD = bsxfun(@times,normals,D); % (#faces x 3) vector from surface to
45  query point*/
46
47              rD0=normals[i]*Daux;
48              rD1=normals[i+mrowsf]*Daux;
49              rD2=normals[i+2*mrowsf]*Daux;
50
51              rD[0]=rD0;
52              rD[1]=rD1;
53             rD[2]=rD2;
54
55              /*P = bsxfun(@minus,point,rD);*/
56
57              P0=qPoint[ip]- rD[0];
58              P1=qPoint[ip+npoints]- rD[1];
59              P2=qPoint[ip+2*npoints]- rD[2];
60
61              Paux[0]=P0;
62              Paux[1]=P1;
63              Paux[2]=P2;
64
65              /*find barycentric coordinates (query point as linear combination of
66     two edges)*/
67
68              rest0_31=r3[i]-r1[i];
69              rest1_31=r3[i+mrowsf]-r1[i+mrowsf];
70              rest2_31=r3[i+2*mrowsf]-r1[i+2*mrowsf];
72
73              r31r31=(rest0_31*rest0_31)+(rest1_31*rest1_31)+(rest2_31*rest2_31);
74
75              /*.....................................................................*/
76
77              rest0_21=r2[i]-r1[i];
78              rest1_21=r2[i+mrowsf]-r1[i+mrowsf];
79              rest2_21=r2[i+2*mrowsf]-r1[i+2*mrowsf];
80
81              r21r21=(rest0_21*rest0_21)+(rest1_21*rest1_21)+(rest2_21*rest2_21);
82
83              /*.....................................................................*/
84
85              aux0_r21r31=rest0_21*rest0_31;
86              aux1_r21r31=rest1_21*rest1_31;
87              aux2_r21r31=rest2_21*rest2_31;
88
89              r21r31=aux0_r21r31 + aux1_r21r31 + aux2_r21r31;
90
91              /*.....................................................................*/
```

```
92              aux0_r31vq=rest0_31*vq[0];
93              aux1_r31vq=rest1_31*vq[1];
94              aux2_r31vq=rest2_31*vq[2];
95
96              r31vq=aux0_r31vq + aux1_r31vq + aux2_r31vq;
97             /*.....................................................................*/
98
99             aux0_r21vq=rest0_21*vq[0];
100            aux1_r21vq=rest1_21*vq[1];
101            aux2_r21vq=rest2_21*vq[2];
102
103            r21vq=aux0_r21vq + aux1_r21vq + aux2_r21vq;
104
105            /*.....................................................................*/
107
108            d=(r31r31*r21r21)-(r21r31*r21r31);
109            bary[0]=((r21r21*r31vq)-(r21r31*r21vq))/d;
110            bary[1]=((r31r31*r21vq)-(r21r31*r31vq))/d;
111            bary[2]= 1- bary[0]-bary[1];
112
113           if  ( ( abs(d)>1e-15) && (bary[0]>0) && (bary[1]>0) && (bary[2]>0) &&
114          (bary[0]<1) && (bary[1]<1) && (bary[2]<1))
115           {
116              /*% find nearest face for query point*/
117
118                if (Daux< mind)
119               {
120                mind=Daux;
121                I=ip;
122
123                P[ip]=Paux[ip];
124                P[ip+npoints]=Paux[ip+npoints];
125                P[ip+2*npoints]=Paux[ip+2*npoints];
126                F[ip]=ip+1;
127                }
128            }
129               else if(ip==0)
130                 {F[ip]=1;
131                  P[ip]=Paux[ip];
132                  P[ip+npoints]=Paux[ip+npoints];
133                  P[ip+2*npoints]=Paux[ip+2*npoints];
134             }
135       }
136        D[ip]=mind;
137       }
138       }
139
140   void mexFunction( int nlhs, mxArray *plhs[],
141                     int nrhs, const mxArray *prhs[])
142   {
143       double  *point, *faces, *normals,*r3,*r1,*r2, *D, *P, *F;
144       size_t mrowsf,npoints;
145       /* check for proper number of arguments */
```

distance_to_surfaces_omp.c

```
146         if(nrhs!=6) {
147          mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs","seven inputs
1478 required.");
149     }
150         if(nlhs!=3) {
151          mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs","three output
152 required.");
153     }
154
155     /*  get dimensions of the input matrix rows  */
156
157     mrowsf=mxGetM(prhs[0]);
158     npoints=mxGetM(prhs[1]);
159
160     /* get dimensions of the input matrix cols*/
161     /*ncols = mxGetN(prhs[0]);
162
163     /* create a pointer to the real data in the input   */
164
165     faces=mxGetPr(prhs[0]);
166     point=mxGetPr(prhs[1]);
167     normals=mxGetPr(prhs[2]);
168     r1=mxGetPr(prhs[3]);
169     r2=mxGetPr(prhs[4]);
170     r3=mxGetPr(prhs[5]);
171
172     /* create the output matrix */
173     plhs[0] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)1,mxREAL);
174     plhs[1] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)3,mxREAL);
175     plhs[2] = mxCreateDoubleMatrix((mwSize)npoints,(mwSize)1,mxREAL);
176
177
178     /* get a pointer to the real data in the output  */
179      D= mxGetPr(plhs[0]);
180      P= mxGetPr(plhs[1]);
181      F= mxGetPr(plhs[2]);
182
183
184     /* call the computational routine */
185
186 distance_to_surfaces_omp(faces,point,normals,r1,r2,r3,D,P,F,mrowsf,npoints);
187 }
```

distance_to_surfaces_CUDA.cu

```
 void __global__ distance_surfaces_CUDA(const double *point,
                                        const double *faces,
                                        const double *vertices,
                                        const double *normals,
                                        double *r1,
                                        double *r2,
                                        double *r3,
                                        double *D,
                                        double *P,
                                        double *F,
                                        int npoints,
                                        int mrowsf
                                       )
{
 int ip=threadIdx.x+blockDim.x*blockIdx.x;

     if (ip<npoints)

{

     double
 vq[3],rD[3],Paux[3],Daux,r31r31,r21r21,r21r31,r31vq,r21vq,d,bary[3],mind;
     double vq1,vq2,vq0,aux0,aux1,aux2,rD0,rD1,rD2,P0,P1,P2,rest0_31,rest1_31,
     rest2_31,rest0_21,rest1_21,rest2_21,aux0_r21r31,aux1_r21r31,aux2_r21r31,aux2_
     r31vq,aux1_r31vq,aux0_r31vq,aux0_r21vq,aux1_r21vq,aux2_r21vq;
    int i;
    mind=100000;

    for (i=0; i<mrowsf; i++)
        {/*vq = bsxfun(@minus,point,r1);  % (#faces x 3)*/

         vq0=point[ip]- r1[i];
         vq1=point[ip+npoints]- r1[i+mrowsf];
         vq2=point[ip+2*npoints]- r1[i+2*mrowsf];

         vq[0]=vq0;
         vq[1]=vq1;
         vq[2]=vq2;

         /*%D = dot(vq,normals,2); % (#faces x 1) distance to surface*/

         aux0+=vq[0]*normals[i];
         aux1+=vq[1]*normals[i+mrowsf];
         aux2+=vq[2]*normals[i+2*mrowsf];

         Daux=aux0+aux1+aux2;

         /*rD = bsxfun(@times,normals,D); % (#faces x 3) vector from surface to
query point*/
         rD0=normals[i]*Daux;
         rD1=normals[i+mrowsf]*Daux;
         rD2=normals[i+2*mrowsf]*Daux;
```

```
54            rD[0]=rD0;
55            rD[1]=rD1;
56            rD[2]=rD2;
57
58            /*P = bsxfun(@minus,point,rD);*/
59
60            P0=point[ip]- rD[0];
61            P1=point[ip+npoints]- rD[1];
62            P2=point[ip+2*npoints]- rD[2];
63
64            Paux[0]=P0;
65            Paux[1]=P1;
66            Paux[2]=P2;
67
68
69            /*find barycentric coordinates (query point as linear combination of two
70    edges)*/
72
73            rest0_31=r3[i]-r1[i];
74            rest1_31=r3[i+mrowsf]-r1[i+mrowsf];
75            rest2_31=r3[i+2*mrowsf]-r1[i+2*mrowsf];
76
77            r31r31=(rest0_31*rest0_31)+(rest1_31*rest1_31)+(rest2_31*rest2_31);
78
79            /*.....................................................................*/
80
81            rest0_21=r2[i]-r1[i];
82            rest1_21=r2[i+mrowsf]-r1[i+mrowsf];
83            rest2_21=r2[i+2*mrowsf]-r1[i+2*mrowsf];
84
85            r21r21=(rest0_21*rest0_21)+(rest1_21*rest1_21)+(rest2_21*rest2_21);
86
87            /*.....................................................................*/
88
89            aux0_r21r31=rest0_21*rest0_31;
90            aux1_r21r31=rest1_21*rest1_31;
91            aux2_r21r31=rest2_21*rest2_31;
92
93            r21r31=aux0_r21r31 + aux1_r21r31 + aux2_r21r31;
94
95            /*.....................................................................*/
96
97            aux0_r31vq=rest0_31*vq[0];
98            aux1_r31vq=rest1_31*vq[1];
99            aux2_r31vq=rest2_31*vq[2];
100
101           r31vq=aux0_r31vq + aux1_r31vq + aux2_r31vq;
102
103           /*.....................................................................*/
104
```

```
105             aux0_r21vq=rest0_21*vq[0];
106             aux1_r21vq=rest1_21*vq[1];
107             aux2_r21vq=rest2_21*vq[2];
108
109             r21vq=aux0_r21vq + aux1_r21vq + aux2_r21vq;
110
111             /*.............................................................*/
112
113             d=(r31r31*r21r21)-(r21r31*r21r31);
114             bary[0]=((r21r21*r31vq)-(r21r31*r21vq))/d;
115             bary[1]=((r31r31*r21vq)-(r21r31*r31vq))/d;
116             bary[2]= 1- bary[0]-bary[1];
117
118             if  ( ( abs(d)>1e-15) && (bary[0]>0) && (bary[1]>0) && (bary[2]>0) &&
119            (bary[0]<1) && (bary[1]<1) && (bary[2]<1))
120            {
121               /*% find nearest face for query point*/
122
123                 if (Daux< mind)
124                {
125                 mind=Daux;
126
127                 P[ip]=Paux[0];
128                 P[ip+npoints]=Paux[1];
129                 P[ip+2*npoints]=Paux[2];
130                 F[ip]=i+1;
131                 }
132             }
133                else if(i==0)
134                  {F[ip]=1;
135                   P[ip]=Paux[0];
136                   P[ip+npoints]=Paux[1];
137                   P[ip+2*npoints]=Paux[2];
138                }
139
140        }
141         D[ip]=mind;
142    }
143    }
144
145
146
```

9.Siglas

CUDA: *Compute Unified Device Architecture*, Arquitectura unificada de dispositivos de computación.

FORTRAN: FORmula TRAslation system, Sistema de traducción de fórmulas.

OpenMP: *Open Multi-Processing*, Multiprocesamiento abierto.

MATLAB: *MATrix LABoratory*, Software con lenguaje de programación propio.

API: *Application Programming Interface*, Interfaz de programación de aplicaciones.

GPU: *Graphics Processing Unit*, Unidad de procesamiento gráfico.

CPU: *Central Processing Unit*, Unidad central de procesamiento.

RAM: *Random Acces Memory,* Memoria de acceso aleatorio.

ITEAM: *Instituto de Telecomunicaciones y Aplicaciones Multimedia.*

10.Referencias

[1]MATLAB, (R2021a), The MathWorks Inc., Natick, Massachusetts, 2021. From https://es.mathworks.com/products/matlab.html

[2] Daniel Frisch (2022). point2trimesh( ) — Distance Between Point and Triangulated Surface (https://www.mathworks.com/matlabcentral/fileexchange/52882-point2trimesh-distance-between-point-and-triangulated-surface), MATLAB Central File Exchange. Retrieved April 6, 2022.

[3] NVIDIA, Vingelmann, P., & Fitzek, F. H. P. (2020). CUDA, release: 10.2.89. Retrieved from https://developer.nvidia.com/cuda-toolkit
Programming Guide :: CUDA Toolkit Documentation (nvidia.com)

[4] NVIDIA CUDA-X, GPU-accelerated libraries for AI and HPC. From https://www.nvidia.com/es-es/technologies/cuda-x/

[5] PARALLEL COMPUTING EXPERIENCES WITH CUDA, from https://uweb.engr.arizona.edu/~ece569a/Readings/GPU_Papers/1.ComputingExperiences.pdf

[6] OpenMP v 4.5 specification (2015). URL http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[7] MEX , Mathworks MATLAB. From https://es.mathworks.com/help/matlab/call-mex-files-1.html

[8] Create mexFunction (C), Mathworks. From https://es.mathworks.com/help/matlab/apiref/mexfunction.html?searchHighlight=mexfunction&s_tid=srchtitle_mexfunction_1

[9] Create mxArray(C), Mathworks. From https://es.mathworks.com/help/matlab/apiref/mxarray.html?searchHighlight=mxarray&s_tid=srchtitle_mxarray_1

[10] NVIDIA AD102. From https://www.techpowerup.com/gpu-specs/nvidia-ad102.g1005